QNX Software Systems Ltd.

175 Terence Matthews Crescent
Kanata, Ontario, Canada K2M 1W8

Tel: 613-591-0931 • 800-363-9001
Fax: 613-591-3579
Internet: info@qnx.com

A Scalable Microkernel POSIX OS for Realtime Systems

Dan Hildebrand, QNX Software Systems Ltd.
danh@qnx.com

*Abstract**

The POSIX API is suitable for a wide range of embedded systems development. Unfortunately, the UNIX heritage of POSIX has generated the impression that a POSIX operating system is too large and burdened with too much overhead to be suitable for small, ROM-based, embedded realtime systems.

This paper explores how the microkernel architecture of the QNX® operating system allows for a POSIX environment suitable for both small embedded systems and large distributed systems. Examples of actual QNX-based systems are included.

*This paper was presented at the Embedded Computer Conference, Santa Clara, California, April, 1993.

301044-v02-p02

Why POSIX for Embedded Realtime Systems?

A common problem with realtime systems development is that each realtime OS tends to come equipped with an API unique unto itself. In the absence of industry standards, this is not an unusual state for a competitive marketplace to evolve into. Informal surveys of the realtime marketplace routinely show that the majority of realtime applications are still using proprietary operating systems. With the advent of the POSIX 1003.4 Draft realtime standard, realtime operating systems from different vendors can provide a compatible API. Why is this a good thing?

Before addressing this question, a brief summary of the POSIX standards applicable to embedded realtime systems is in order. The POSIX effort is split into a number of groups, each focused on a specific aspect of operating system functionality.

The POSIX 1003.1 And 1003.2 Standards

The POSIX 1003.1 standard defines the API for process management, device I/O, and filesystem I/O. This encompasses what might be described as the base functionality of a UNIX operating system, 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 in terms of standard I/O. The full details of the 1003.1 standard can be found in [1]. The POSIX 1003.2 standard goes on to define the syntax supported by the shell and utilities, the utility command-line options, and the format of the output from those utilities. Compliance with this standard is important to allow developers familiar with UNIX systems to be immediately "comfortable" with the development environment. A complete listing of the utilities included in 1003.2 can be found in [2].

POSIX 1003.4 Draft Realtime Standard

The 1003.4 Draft realtime standard [3] defines a set of realtime extensions to the base 1003.1 standard. These extensions consist of such things as semaphores, prioritized process scheduling, realtime extensions to signals, high-resolution timer control, enhanced interprocess communication (IPC) primitives, synchronous and asynchronous I/O, and realtime contiguous file support. See the draft standard for more complete details.

Currently the 1003.4 standard is at Draft 13, and will likely emerge as a full standard sometime in 1993. Once 1003.4 becomes a standard, a number of operating systems are likely to adopt some or all of this standard, since the services defined are generally useful to many classes of applications. Even general user interface applications can benefit from the application of a better realtime API within the OS.

Associated with 1003.4 are two additional standards: the 1003.4a draft standard threads extension for POSIX and the 1003.4b draft standard for extra realtime interfaces. The 1003.4b standard attempts to define a set of extensions to the 1003.4 base standard for such items as application-level interrupt handling, spawn (a combined *fork()*/*exec()*), rate-monotonic scheduling, and CPU time monitoring of thread execution.

POSIX 1003.13: Draft Standard for Application Environment Profiles (AEP)

Early in the definition of the POSIX 1003.4 Draft realtime standard it became apparent that not all realtime systems, particularly embedded systems with smaller processor and memory resources, could physically support the entire POSIX 1003.4 functionality, especially since the 1003.4 realtime standard presupposes the presence of the complete 1003.1 standard. At the same

time, larger realtime systems need all these standards to coexist. To address this range of application needs, the 1003.13 Application Environment Profile draft standard was created [4].

This standard defines four levels of functionality for POSIX 1003.4 implementations. Since some of the hardware intended for embedded applications may not be capable of fully supporting all of POSIX 1003.1 or 1003.4, this standard exists to define specific subsets of POSIX such that reduced-functionality implementations can exist for less capable hardware, as is often found in embedded systems for reasons of cost, size, power consumption, etc. For example, an embedded system without an MMU is unable to support the POSIX 1003.1 *fork()* call.

The four profiles defined by the 1003.13 standard are:

- a minimal realtime system
- a realtime controller
- a dedicated realtime system
- a multi-purpose realtime system

The minimal realtime system is typically a system without a filesystem or user interaction, where the I/O devices usually take the form of parallel ports or analog I/O devices. The realtime controller adds an optional filesystem, with serial tty device control and support for POSIX.1 signals. The dedicated realtime system is an extension of the minimal realtime system, adding a simple, non-hierarchical filesystem, tty device drivers, and support for multiple processes. The multi-purpose realtime system includes all of POSIX.1 and POSIX.4, and can be used to run a mix of realtime and non-realtime processes.

The Completeness of the POSIX Standards

To support the porting of common UNIX source code, the POSIX standard serves as a good starting point, but still leaves a number of standard UNIX services unspecified. Since most POSIX implementations to date exist on UNIX systems—which provide a UNIX superset of POSIX capabilities—this limitation has not been a significant problem. However, for non-UNIX-derived POSIX implementations attempting to support UNIX-derived application source code, it becomes necessary to assess the services provided by UNIX at large and to provide implementations of the common services provided by UNIX but not defined by POSIX. For example, curses, terminfo/termcap, *readv()*/*writev()*, *ioctl()*, *select()* and other commonly used Berkeley, SVR4, and SVR3 services are left undefined by POSIX, but are commonly used by UNIX applications.

For this reason, any OS providing only the “vanilla” POSIX API services will find that porting UNIX source is still problematic. It will be interesting to observe the industry acceptance of the POSIX subsystem within the upcoming release of Windows NT, given its purportedly strict “POSIX 1003.1 and nothing more” implementation. Within the QNX environment, we have found it useful to add many of the common UNIX facilities in order to make the environment suitable for generic ports of UNIX source code. With these added facilities, most UNIX source code ports to QNX with only a modest investment of effort. Note that source code utilizing these features will not necessarily run on other strictly POSIX-compliant systems, but general usability standards dictate the adoption of these UNIX services all the same.

Posix Standards and Realtime Embedded Systems

Returning to the original question of why a POSIX standard API is useful for embedded systems development, the application of standards would unify the variety of APIs provided by realtime OS vendors. The common reasons for why standards are beneficial apply as well to the

embedded marketplace, although the requirements for additional performance and low-level flexibility present new challenges for the development of these standards.

Some of the reasons to encourage standards for realtime operating systems include:

1) Source code portability. A common goal for most language standards, such as with X3J11 ANSI C, is to provide portability of program source code between different processors and operating systems. In order to address operating system portability, ANSI C defined a number of I/O services, but the ANSI-C-defined services alone are not adequate for more complex applications. In order to achieve better OS portability for applications, the POSIX standards attempt to provide a richer and more complete API for the underlying OS.

The need for better application portability is especially relevant in the realtime market, given the wide variety of processors and operating systems commonly used. However, an additional problem with realtime development is that often large portions of an application are implemented in assembly language. Given the routine need for unusual interface hardware, the assembly code portion of an application can be substantial, purely because of the device drivers. There is also a perception that assembly code necessarily means greater performance for the final application, which is always of paramount concern for realtime systems.

The disadvantage of using assembly code for better performance, or for greater flexibility for the implementation of device drivers, is that with newer processors the exact instruction sequencing needed to achieve maximum performance is no longer obvious. Newer compilers that perform instruction scheduling are becoming more common, making a case for the view that the compiler can actually do a better job than the assembly programmer for these processors. As a result, applications expressed purely in a high-level language can take advantage of a new processor simply by recompiling.

For example, for best performance the Intel Pentium processor requires that the instruction stream be appropriately sequenced such that the dual pipelines are both kept “busy” as often as possible. To achieve this, hand-written 486 assembly code would need a substantial rewrite, while C source requires only a recompile with the appropriate compiler.

In our own development work with the QNX operating system, the transition to the optimizing Watcom C compiler has resulted in the replacement of several hundred lines of assembly code in the device drivers and OS kernel with fairly “vanilla” C source, leaving only a few tens of lines written in assembler. Performance was comparable or superior when the conversion was complete. Within QNX we also implemented high-level interrupt access services, allowing device drivers to be entirely implemented in C. As a result, QNX was able to immediately take advantage of the Intel Pentium processor by using the Watcom v9.5 C Compiler, which can generate Pentium-optimized code.

As the POSIX working groups further define the 1003.4b and 1003.13 draft standards, a common API for the implementation of interrupt handlers and device drivers will also emerge. This will further increase the functionality of applications written entirely in high-level languages, decreasing the need for assembly code and providing a greater degree of portability for realtime applications.

2) Processor independence. The use of high-level languages and a common OS API allow greater flexibility of processor choice, since strict assembly-level compatibility with previously developed systems is no longer needed. The processor for a given project can be chosen on the basis of development tool availability, performance, cost, and other relevant factors, rather than on the basis of an existing processor-specific and OS-specific source code body to draw upon. This allows companies to free themselves of traditional or historical dependence on single-processor families and to follow the marketplace as various processor families emerge.

3) Portability of development staff. With a common API for realtime operating systems, programmers experienced with one realtime OS can directly apply their skills to other projects, using other processors and operating systems. In addition, the skills of programmers using POSIX or UNIX systems become applicable to realtime systems, since the non-realtime portion of the realtime OS's API will already be familiar to UNIX programmers. From a staffing perspective, this allows members of a development team to "float" between projects more readily and simplifies the hiring of additional staff.

4) Development environment: native vs cross development. A workstation running a POSIX OS can, with the addition of I/O hardware similar to the target embedded runtime system, become a functional superset of that system. As a result, more of the system can be developed on the workstation before having to resort to a "cross development" environment. If the target hardware platform is one of the huge variety of PC-compatible systems (Ampro Littleboard, EISA bus, Microchannel, Multibus I & II, PC/104, STD and STD-32 bus, VME bus, passive backplane ISA bus, VESA local bus, etc.) an 80x86-based PC can be used as a development platform that is functionally identical to the runtime platform, regardless of what form the final packaging takes. This allows developers to use generic (and inexpensive) Intel-based PCs as reasonably powerful development systems that are functionally identical to the target embedded system. The variety of packaging options for the PC architecture also ensures that a cost-effective hardware platform can meet the requirements of performance, robustness, power consumption, etc. With less dependence on cross-development environments, R&D time and time-to-market can be decreased.

Scaling Down POSIX for Small Embedded Systems (POSIX != UNIX)

Since the POSIX standards are primarily adopted from existing UNIX practice, POSIX and UNIX systems share many common attributes. Unfortunately, the traditional UNIX monolithic kernel architecture has resulted in operating systems that generally tend to be too large and non-realtime for embedded realtime systems. By association, POSIX systems unfairly share some of this reputation. In addition, UNIX systems tend to rely heavily on the presence of a filesystem, something many embedded systems deliberately omit.

Although the POSIX standards have their root in existing UNIX practice, the POSIX working groups carefully defined the standard in terms of interface, *not* implementation. The precision of specification within the standards, as well as the availability of POSIX test suites, allow non-traditional OS architectures to provide a certifiable POSIX API without having to "bundle in" a traditional UNIX kernel to maintain the subtleties of compatibility that might otherwise be lost if the original UNIX kernel source were not present. As a result, OS architectures more suited to embedded realtime systems can now provide a compatible implementation of the services defined by the POSIX API standards.

A Microkernel Operating System Architecture

A microkernel operating system architecture is one approach to providing a POSIX API implemented over an OS more suitable to embedded realtime systems. A microkernel operating system consists of a relatively small kernel providing minimal services, surrounded by a set of processes providing higher level OS services such as file I/O, device I/O, and networking.

Although the name "microkernel" implies that the point of the architecture is to make the kernel "small", the term refers more to the partitioning of the OS than to the size of the kernel. For example, at the high end of the scale, Mach has a "micro" kernel in the order of 100 to 200K, while QNX provides a microkernel of only 8K (which fits into the 486 on-chip processor cache quite nicely). These kernel sizes are important, but the architecture is the real issue at hand.

A good analogy to clarify this microkernel vs monolithic kernel comparison is the RISC vs CISC processor discussion. The literal meaning of the term RISC (reduced instruction set computer) is not very meaningful or useful, since the so-called RISC RS6000 processor has *more* instructions than the aptly named CISC Intel 486! But if you apply the reasoning that RISC is a design approach to allocating silicon only to where it best improves the processor's performance, flexibility, parallelism, or other attribute (thereby forcing features not critical to that desired functionality to be implemented in some other manner), then you end up with a more interesting (and more useful) definition for RISC. The correct tradeoffs can result in better performance for the final system.

When applying this approach to a microkernel OS, the key point is to implement within the microkernel only those functions that are most crucial to the OS's performance, flexibility (or any other desired attribute), and then implement the remaining functionality by other means (such as via processes managed by that microkernel). When the kernel is defined in terms of its most essential functionality, it can best deliver those characteristics because of the reduced problem scope. The challenge remains for the OS implementor to provide the rest of the OS services in such a way as to not hamper the gains made in kernel performance as other functionality is added.

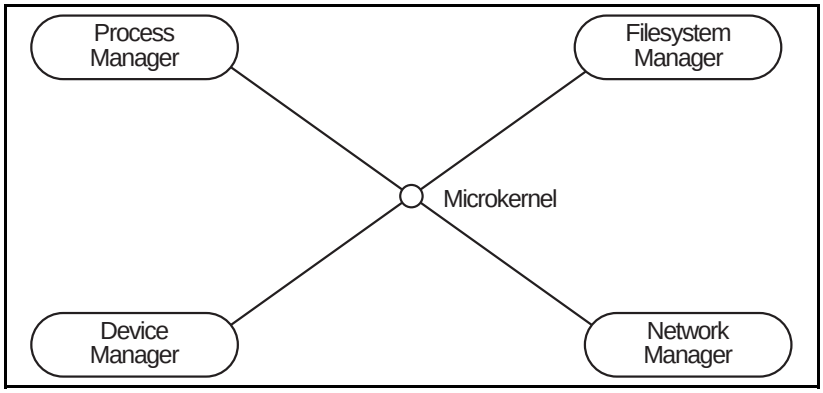

Figure 1. A generic microkernel surrounded by a team of cooperating processes.

For embedded realtime applications, the implementation of a dedicated microkernel with high-performance context switch and interrupt latency characteristics is clearly a worthwhile goal. The microkernel aspects of the kernel implementation then allow higher-level services to be added without diminishing the performance of those microkernel services. If the context switching and message-passing performance are sufficiently quick, the add-on processes can provide their services as competently as the monolithic kernel equivalent, while delivering the greater architectural flexibility required by the embedded systems developer.

Properly implemented, a microkernel in its minimal state can be suitable for tiny, embedded realtime applications. Or it can be incrementally scaled up into as complete a realtime UNIX/POSIX system as is needed. A system could use an "all options included" microkernel OS to provide a self-hosted development environment, and later remove the extra functionality to implement the final runtime system.

The POSIX 1003.13 APE draft standard with its four levels of functionality clearly reflects the market demand for various subsets of the POSIX realtime standard. Different classes of embedded systems need various levels of POSIX functionality, but having to burden small-scale systems with unnecessary high-level functionality is far from cost-effective for high-volume

embedded systems. Other constraints such as limited memory, small (or non-existent) disk space, and perhaps limited processor functionality make the need for subsets of POSIX functionality even more apparent. Rather than use different operating systems to deliver the various levels of 1003.13-defined Application Environment Profiles, a single microkernel operating system, with its inherently high degree of configurability, can be used to deliver many, if not all, of these levels.

Modularity and Maintainability

A problem that often arises with the maintenance of a monolithic kernel operating system is that all kernel code runs in a common, shared address space. The danger that one portion of the kernel might corrupt something else is very real and must be taken into account every time new drivers are linked into the kernel. The approach of microkernels such as QNX is to explicitly define the interfaces between the components that make up the OS in such a way that each resource manager, just like user processes, runs in its own memory-protected space, and all communication between the OS modules is through standard IPC services. As a result, errors introduced by one resource manager will be constrained to that subsystem and will not corrupt other, unrelated resource managers in the system. This modularity is also the key to being able to configure different versions of the OS, since extra functionality can be readily omitted without reducing the viability of the rest of the system.

Microkernel Myths

Two criticisms are commonly (and erroneously) made of message-passing, microkernel operating systems. One is that microkernel OSs are slower than monolithic-kernel OSs because of the additional context switching between the processes that make up the OS. The second criticism is that a message-passing OS has more overhead than an OS based on kernel calls.

The first criticism becomes invalid if the microkernel can provide sufficient performance that the context-switch time is insignificant against the overall interval required to perform the OS service request. Having minimized the context-switch overhead, the returned gain is the greater degree of parallelism in the services performed by the OS in separate processes. These processes can be scheduled at client-driven priorities, such that high-priority user level processes can preempt OS work done on behalf of other, lower-priority user processes. In general, the lower overhead of the microkernel implementation allows overall system performance to exceed that of a monolithic kernel implementation [5].

The second criticism regarding message-passing overhead is also easily refuted. Since a microkernel can use multipart message passing to allow the kernel to move data from one process to another, redundant copying to build contiguous messages is not needed. In addition, most messages passed tend to be very tiny (on the order of 20 bytes). Given the rate at which processors can block-move memory, the copying of a 20-byte message is not a significant overhead. In the event of a large message being copied, as with disk I/O, the multipart message services allow the kernel to copy the disk block without any redundant copies, exactly as would be done in a monolithic kernel. Large copies done by the microkernel can also be preempted by higher-priority requests made by other processes.

It has been suggested that it is more efficient to pass pointers to messages rather than the message itself. However, the advantage gained by passing the message is that semaphores are not required to govern which process “owns” the message. Semaphore processing can easily become more burdensome than copying small messages. Another advantage of the message pass is that it distinctly separates the address spaces of the user and provider of a system service, conveniently allowing them to run on different network nodes with full transparency of operation and full memory protection between processes.

The QNX Microkernel

Architecturally, QNX is representative of the "classic" microkernel architecture, with architectural extensions for network-transparent distributed processing. The cooperating processes surrounding the microkernel provide various high-level OS services, but are no different from user processes except in their privilege level. Sufficiently privileged user processes can also provide OS services.

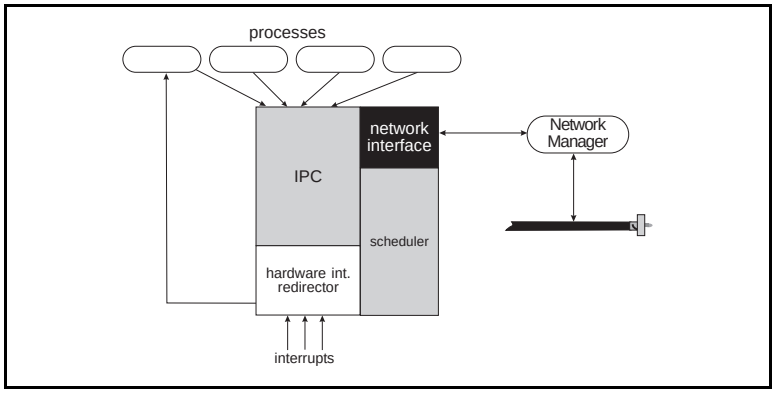

Figure 2. The QNX microkernel architecture allows for extended functionality without sacrificing realtime performance.

The QNX microkernel is comparable to a small realtime executive in terms of the services provided and the performance of those services, but differs from many realtime executives in that it can also host additional processes that extend the base functionality of the system. Each process added provides that portion of the POSIX API (e.g. filesystem, device I/O, etc.). With all of the optional processes present, a fully compliant POSIX API is available to applications.

The 8K microkernel implements three basic services with 14 kernel calls:

- interprocess communication (local or network-directed message passing)
- process scheduling
- interrupt dispatching

Unlike the optional server processes, the kernel itself is never scheduled for execution, being invoked only through kernel calls from either a process or a hardware interrupt.

The message-passing IPC facilities provided by the microkernel are blocking versions of *Send()*, *Receive()*, and *Reply()*. For example, a process that does a *Send()* to another process will be blocked until the target process does a *Receive()*, processes the message, and does a *Reply()*. If a process executes a *Receive()* without a message pending, it will block until another process executes a *Send()*. QNX attaches no special meaning to the content of a message—the data in a message has meaning for the sender of the message and for its receiver, but for no one else. However, "well-defined" message types are provided (and documented) so that user-written processes can substitute for or augment the system-supplied processes.

Since these message-passing facilities block the sending or receiving process, they also synchronize the execution of communicating processes. This "minimalist" approach to IPC allows the kernel to implement IPC as efficiently as possible. QNX copies a message directly from the address space of one process process to another without intermediate buffering, so message-delivery performance approaches the memory bandwidth of the underlying hardware. All system services are built upon these high-performance message-passing services. When other

forms of IPC are required, they can be efficiently implemented by using these basic message-passing services (e.g. pipes, message queues, etc.), and the performance of the base IPC services proves more than adequate to compete with other kernel-based queuing systems [5].

With QNX, processes can request that messages be delivered in order of priority, rather than in a FIFO order. Server processes can request message-driven priority—they execute at the priority of the process they are serving, and yet automatically have their priority raised when a higher-priority process blocks on the already-busy server. As a result, a low-priority process cannot preempt a higher-priority process by invoking the services of an even higher-priority server. Simply put, this message-driven priority mechanism avoids priority-inversion problems. These process-scheduling primitives conform to the POSIX 1003.4 (realtime) draft specification. QNX provides fully preemptive, prioritized context switching with round-robin, FIFO, and adaptive scheduling.

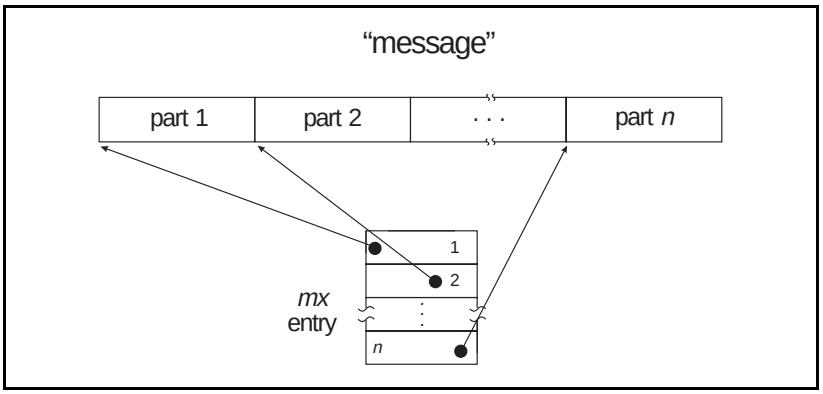

Figure 3. Multipart messages can be specified with an MX control structure.

The messaging primitives in QNX also support multi-part messaging, so that a message delivered from one process to another need not occupy a single, contiguous area in memory. Instead, both the sending and receiving processes can specify an MX table that indicates where the sending and receiving message fragments exist in memory. This allows 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, 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 to this concept would be that of a scatter-gather DMA facility.

The Process Manager

In addition to the microkernel, the only other required component to build a functional QNX OS is the process manager (**`Proc`**). **`Proc`** provides services such as process creation, process accounting, memory management, process environment inheritance (both locally and for network-remote processes) and pathname space management. A minimal embedded QNX system can be built from a microkernel, a process manager, and one or more application processes. The OS component of this minimal system (including a shared library) is roughly 80K. The filesystem or device I/O system is optional. First-level pathname space management (referred to as prefix space) is done by **`Proc`** because the filesystem process is purely optional under QNX.

Until resource manager processes begin execution, **Proc** owns the entire pathname space, including the root and everything beneath it. Without any resource managers present to provide services, this is equivalent to an empty filesystem.

Proc allows resource managers through a standard API to adopt a portion of the prefix namespace as a "domain of authority" to administer. As other resource managers adopt their respective domains of authority, **Proc** becomes responsible for maintaining the prefix tree to track which processes own various portions of the pathname space. This approach to pathname space management is what allows QNX to preserve the POSIX semantics for device and file access, while making the presence of those services optional for reduced-size embedded systems.

With **Fsys** (the filesystem manager) and **Dev** (the device manager) running, the prefix tree would look like the following:

/	Disk-based filesystem (**Fsys**)
/dev	Character device system (**Dev**)
/dev/hd0	Raw disk volume (**Fsys**)
/dev/null	Null device (**Dev**)

When a process opens a file, the POSIX-compliant *open()* library routine first sends the filename to **Proc**, where the pathname is compared against the prefix tree in order to determine which resource manager should be sent the *open()* message. In the case of partially overlapping domains of authority, the longest match to the pathname would win. For example, if **/dev/tty0** were opened, the longest match would occur on **/dev**, causing the open to be directed to **Dev**. The pathname **/usr/fred** would match against **/**, directing the open to **Fsys**.

The process manager on each computer in a network maintains its own prefix tree and may present identical or different views of the network-wide pathname space to processes on each node. Pathnames that start with a **/** are applied against the prefix tree on that node.

Network-unique names are also available to allow applications to specify the absolute location of resources within the network-wide pathname space. Through the use of prefix aliasing, portions of the namespace can be mapped to resource managers on other network nodes. For example, a diskless workstation that booted from the LAN and wanted to have its filesystem root on another node could alias the root of its filesystem to a remote filesystem process.

With this alias in place, *open()* calls to **/dev** would still map to the local **Dev** process for control of local devices, but *open()* calls for files would result in open messages being resolved by the prefix-mapping table on the previously specified node (which would usually direct file opens to the **Fsys** process on that node). As a result, processes anywhere on the network can access all of the network filesystem resources within a single directory tree connected to a common filesystem root **/**. Alternatively, by using network-absolute pathnames, the network pathname space can be manipulated as a collection of individual root filesystems (one tree or a forest of trees).

By implementing individual domains of authority within the conventional filename paradigm, portions of the overall functionality of the OS can be implemented in a runtime-optional manner. Since resource managers live outside the kernel space and are not fundamentally different from user processes, they can be added or removed dynamically at runtime, without requiring the kernel to be relinked to contain different levels of functionality.

The network transparency of this namespace allows remote execution of processes to be logically equivalent to execution on a local processor. The individually administered pathname spaces blend seamlessly, and there are no "surprises" in how the namespace behaves. Inheritance of the entire parent process environment, including open file descriptors, environment variables, and the current working directory, is done such that IPC and file I/O operate in accordance with the POSIX 1003.1 specification, in spite of the network-distributed runtime context.

The Filesystem Manager

Fsys is the resource manager that provides a POSIX-compliant filesystem for the QNX environment. It implements a disk structure that uses a bitmap for free space allocation, and an array-of-extents approach to organizing the data on disk. This approach allows the filesystem to deliver application-level performance nearly equal to the raw transfer rate of the disk hardware. A special "tiny" filesystem is available for small-scale embedded systems that don't need a fully implemented POSIX filesystem. Filesystems for DOS and ISO-9660 CD-ROM media also exist.

Fsys performs synchronous writes to disk for data structures critical to filesystem integrity, allowing the disk system to survive unexpected power outages. Special tags embedded in on-disk data structures also allow the filesystem to be easily rebuilt in the event of a catastrophic failure.

The multithreaded architecture of **Fsys** allows it to deal with multiple requests in parallel, such that ramdisk and cache I/O can occur while other threads are blocked, waiting for physical I/O to occur. This parallelism extends down into the driver as well. If a device can support multiple pending I/O requests, then the requests can be serviced by the device in whatever order is appropriate to the attached hardware.

The Device Manager

The device manager (**Dev**) provides POSIX-compliant device control with some extensions suitable for realtime communications. Both **Dev** and **Fsys** can be dynamically started and their device drivers attached. Later, these processes can be removed from memory if no longer needed.

Multipart messaging primitives allow **Dev** to direct the *write()* done by an application right into a ring buffer managed by an interrupt handler. Since writing to a ring buffer can require the data to be mapped into physically disjoint, but logically contiguous, memory regions, an MX table with three entries can describe the header and the two physically disjoint sections of the ring buffer. For the *read()* case, the data flow from a device goes directly from the driver into the ring buffer, and from the ring buffer into the application's *read()* buffer without redundant copying to build contiguous messages.

Interrupt Handlers

Rather than insist that device-driver interrupt handlers live only in the kernel space, QNX provides a system call that allows user processes to connect a handler within a sufficiently privileged user process to a particular interrupt vector within the kernel. This system call allows user processes written in high-level languages to easily implement interrupt handlers without resorting to assembly code. The connected handler will then be called by the kernel in response to physical interrupts. By residing within the user process, the handler has full access to the address space of the process for the purpose of responding to the interrupt. Once the handler has run, it can either wake up the process it shares code with or simply return to the kernel.

The benefit of interrupt handlers that reside outside the kernel is that the user can dynamically add and remove them (and the device drivers that contain them) from a running system. The first-level interrupt handling done by the kernel also takes care of nested and shared interrupts without imposing the hardware-dependent details on user-written interrupt handlers. External interrupt handlers thus allow the microkernel's resource manager to match the level of performance that resource management within a monolithic kernel could provide. This also allows the device drivers needed for embedded systems to be easily implemented within user-level processes.

An advantage of having device drivers reside within user-level processes is that developing extensions to the OS is not functionally different from developing user-level processes. In fact,

the development approach used in house is to execute experimental resource managers under the control of the full-screen, source-level debugger. This allows for the debugging of OS services such as a new **Fsys** process without having to resort to low-functionality tools such as the kernel debuggers typically used to debug kernel-linked extensions for monolithic operating systems.

The Network Manager: Distributed and Multiprocessing

Although once not common in realtime executives intended for embedded realtime systems, the ability to network embedded systems is becoming increasingly important. Many embedded applications require a team of processors distributed within a small physical space, working in concert as a small-scale distributed realtime system. Other applications require a large collection of distributed processors located throughout a factory. Connectivity with enterprise-wide networks is also important for purposes of general information flow throughout a company.

The QNX network manager (**Net**) is directly connected to the microkernel message-redirection services. When the microkernel needs to pass a message from a local process to a process on another node, it passes the message through the **Net** manager, which will use the device drivers attached to it to direct the message onto the appropriate LAN interface. Similarly, **Net** can receive messages from remote microkernels and inject those messages into the local microkernel, to be routed to the target process.

Since all system services are accomplished via message passing through the microkernel, and since the **Net** process serves to merge the microkernels on the network into a single microkernel, the result is a network of machines that behaves like a single, logical computer. Any services provided in higher architectural layers of the OS are transparently accessible to all processes on the network. This extends the scalability of the microkernel from tiny, ROM-based systems up to systems comprising hundreds of distributed processors merged into a single logical machine.

Just as **Fsys** and **Dev** (which can be started and stopped from the command line) have their own family of drivers, **Net** also has a family of drivers and supports the connection of multiple network drivers to **Net**. If **Net** discovers that more than one of the network drivers provides connectivity to the same node, it will load-balance the traffic between the drivers. The use of multiple network paths between nodes provides both better throughput and network fault-tolerance. Application-level changes are not needed to take advantage of this fault-tolerance, since the support exists internal to the network manager itself.

The **Net** process implements a QNX-specific lightweight LAN protocol intended to deliver the full performance of the LAN interface to the application layer. Since the generalities of heterogeneous LAN protocols like TCP/IP are not needed for QNX-to-QNX communications, this protocol provides absolute minimal overhead for QNX network transactions. Low overhead at this level is necessary, since the intent of the QNX LAN is to transform the QNX nodes on the LAN into a single, logical QNX computer. In the event that the LAN link is a bus (a VLAN: Very Local Area Network), such as STD-32 or Multibus II, low-overhead protocols are necessary to realize an appreciable portion of the data-transfer capabilities of the hardware.

In the event that communication between a QNX and a non-QNX node is required, **Net** also makes available a raw packet interface that other processes can attach to in order to provide alternate LAN protocols for other QNX processes. Both TCP/IP and DOS LAN services are available in this manner. To the external world, the cluster of QNX nodes on the LAN appears to be a single, logical machine (consisting of several processors). Since the QNX lightweight LAN protocol uses standard packet headers, it peacefully coexists with other LAN protocols on the same cable. For a VLAN system, a backplane full of processors can communicate internally via the QNX protocol and externally via TCP/IP and/or DOS LAN protocols.

Satellite Embedded Processors

By taking a minimal QNX system (microkernel and **Proc**) and adding the **Net** process and a network driver, a tiny, embedded system can become part of the "logical computer" formed by the other nodes on the network. Any process on the embedded system can access the resources of any other node on the network as if it were a local service. As a result, the embedded system will inherit the disk, graphics, database, and other resources of the entire network, and the rest of the network will be able to communicate with the processes on the embedded system. By adding multiple LAN links between nodes, a very high-bandwidth, highly reliable distributed control system can be built. These networked, embedded processors can be quite compact and inexpensive, given that network interfaces such as serial lines (RS-232, RS-422, etc.) or single-chip network interface controllers can be used.

Fault-tolerance and Network Determinism

An advantage of having multiple LAN links connecting network nodes together is that the reliability and determinism of network communications can be preserved by providing surplus LAN bandwidth or alternate LAN protocols. For example, by adding an Arcnet link to an existing Ethernet link between larger systems, those systems would experience the high-burst data rate of the Ethernet when available (1 Mbyte/second), but in the event that the Ethernet is experiencing high-traffic loads and packet collisions due to its CSMA nature, the Arcnet connection can provide a better guaranteed "worst case" data rate while Ethernet is under load. Additional point-to-point LAN links can be added between mission-critical or communications-intensive nodes to provide private data paths as well, further guaranteeing network determinism and fault-tolerance.

Even with significantly slower secondary LAN links, such as RS-232 serial lines, a distributed realtime system can respond intelligently in the event of a partial LAN failure. Although the presence of multiple LAN links is transparent to the application level, a reduction in LAN bandwidth may prevent the realtime application from operating correctly. In the event of a partial LAN failure, the application can use the remaining connectivity bandwidth to intelligently shut down the system and provide diagnostic network services until the additional LAN links can be brought up again.

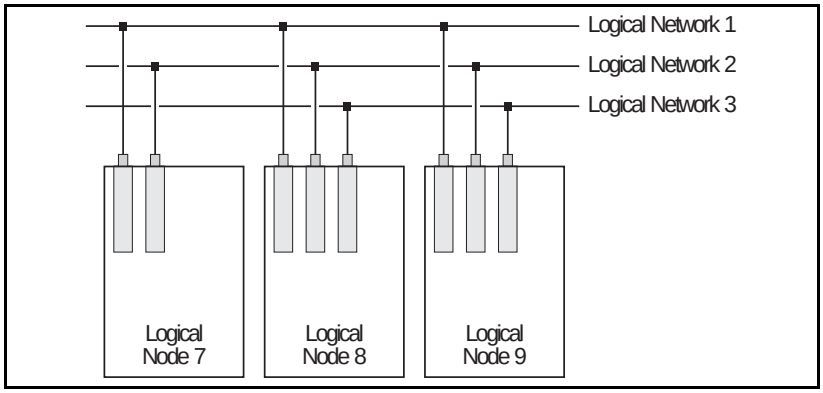

Figure 4. Multiple LAN links between nodes can enhance throughput, reliability, and determinism.

Performance Summary

Within the QNX microkernel, sufficient architectural optimization has been accomplished to achieve a 12μsec context switch on a 33MHz 486 processor. For virtually all system calls, the processing done to satisfy the system call (disk I/O, etc.) outweighs the few microseconds consumed by the context switch. Having achieved this level of performance, the additional architectural overhead imposed by splitting the OS into team of messaging, cooperating processes becomes negligible. However, the resulting gains of greater parallelism within the OS produces a better performing operating system in general.[5]

Here is an example of the performance profile seen by application-level processes for QNX on a 33MHz 486 with 8K bytes of on-chip cache, no external processor cache, an 8MHz ISA bus, an NE2000 Ethernet card, a 16450 UART, and an Adaptec 1542 SCSI disk controller card driving a 1Gbyte Micropolis SCSI hard disk:

Context switch:	12 μsec
Interrupt latency:	6 μsec
Disk:	1.8 Mbytes/sec (equivalent to the actual platter speed)
Network:	1050 Kbytes/sec (1.05 Mbytes/sec)
Serial I/O:	115Kbaud

These performance numbers bear out the assertion that message passing can certainly deliver the full I/O performance of the hardware to application-level processes. The low overhead means that either higher-end applications can be addressed with a given level of hardware, or that the hardware can be reduced in cost, since lower-performance hardware may also be adequate for the realtime application.

Product Scaling

Several advantages can be realized by using a microkernel OS that can be scaled to address applications ranging from small, ROM-based embedded realtime systems, to large, several-hundred processor distributed systems. One advantage is that a common set of development tools and application code can be applied to a range of systems. Different products within a product line can be implemented by adding or removing components of the OS and components of the application to configure various degrees of functionality, thereby achieving products with varying price, performance, functionality—all from a single development effort. For larger-scale systems installed in the field, the distributed processing achieved by the network-merged microkernels can be extended via additional network nodes for additional functionality or better fault-tolerance through the assignment of network nodes to act in a “hot-standby” capacity.

Small-scale system example: elevator control

By placing only the QNX microkernel and process manager into the ROM of a system based on one of the embedded-PC hardware standards, Millar Elevator Industries Inc. [6] is able to implement a very cost-effective, embedded elevator car controller. As an example where networking becomes a natural extension of an embedded system, larger buildings can add a dedicated computer to act as an elevator scheduler. By adding the **`Net`** process to the elevator car controllers, these embedded systems become a logical extension of the computing environment on the scheduling machine. Processes on the elevator car controllers can use resources available on the scheduler and on other car controllers. The embedded car controllers can network-boot from the scheduler machine, but as ROM-based embedded systems, they can also run autonomously in the event of a LAN or central computer failure.

Large-scale system example: nuclear reactor monitoring

A QNX-based nuclear reactor monitoring system is being installed in a U.S.-designed nuclear power plant in Slovenia. Built by Maritime Nuclear, a branch office of Atomic Energy of Canada Ltd. [7], the system consists of over thirty VME and passive backplane ISA-based computers inter-connected with a dual-redundant Arcnet network. Each computer is synchronized by a master time source to within ± 10μsec, ensuring that network-distributed data acquisition and control processes are synchronized.

The mandatory fault-tolerance for this application is achieved by organizing the computers into dual-redundant pairs. Each computer within a dual-redundant pair performs data acquisition and control functions in parallel, providing complete functional redundancy. The system design is tolerant to all single-point failures and many combinations thereof without loss of system-level functionality. Each computer runs a collection of processes as dictated by the system function it performs. Data acquisition and control machines make little use of file and console I/O services, while operator interface machines rely on additional services, such as windowing systems. Computers dedicated to maintenance and configuration provide full filesystems, including backup facilities. Despite the physical distribution of machines and the fault-tolerant setup, the entire computing cluster operates as a single, logical computing resource.

DOS Compatibility

Although microkernel operating systems are architecturally very different from DOS, this does not preclude the ability to support DOS applications. Since QNX runs on PC hardware, and since DOS applications are abundant for that platform, support for DOS programs is very much in demand. QNX's **Rundos** program allows most DOS applications, including MS Windows in standard mode, to run within QNX. The execution environment provided for MS Windows allows several megabytes to be given to MS Windows, enabling most MS Windows applications to run well. DOS applications running under Rundos use the native QNX network-distributed filesystem, which provides DOS-compatible network file locking so that DOS network applications can share files appropriately. A facility for mounting DOS-native filesystems into QNX also allows both QNX and DOS applications to read and write DOS media, on any network node.

With QNX's high-performance networking and filesystems, DOS applications frequently demonstrate better performance under QNX than under native DOS [8]. DOS LAN applications also benefit from the fault-tolerant nature of the multiply-redundant networking provided by **`Net`**.

Conclusions

The successful implementation of various embedded realtime systems using microkernel operating systems such as QNX clearly proves the viability of POSIX-based microkernels for embedded applications. As the POSIX standards advance further, embedded system developers will have an ever greater variety of tools for developing high-performance, highly portable applications.

References

[1] *POSIX Std 1003.1-1988*.

[2] *POSIX Std 1003.2-1992.*

[3] *POSIX Std 1003.4-199x (Draft 13, September 1992).*

[4] *POSIX Std 1003.13-199x (Draft 5, February 1992).*

[5] Dan Hildebrand, *An Architectural Overview of QNX*. The Proceedings of the Usenix Workshop on Micro-Kernels & Other Kernel Architectures, Seattle, April, 1992.

[6] Millar Elevator Industries Inc., 620 12th Avenue, New York, NY 10036.

[7] Atomic Energy of Canada Ltd., Maritime Nuclear, 64 Alison Blvd., Fredericton, New Brunswick E3B 5W6.

[8] Tim Parker's review of QNX in "Off the Shelf," *UNIX Review*, March, 1993, pp. 53-57.

QNX Software Systems Ltd.

175 Terence Matthews Crescent
Kanata, Ontario, Canada K2M 1W8

Tel: 613-591-0931 • 800-363-9001
Fax: 613-591-3579
Internet: info@qnx.com

Printed in Canada