Build a more reliable world™

QNX Software Systems Ltd.
175 Terence Matthews Crescent
Ottawa, Ontario, Canada, K2M 1W8
Voice: +1 613 591-0931 1 800 676-0566
Fax: +1 613 591-3579
Email: info@qnx.com
Web: www.qnx.com

Real Time or Real Linux? A Realistic Alternative

Paul N. Leroux
Technology Analyst
QNX Software Systems Ltd.
paull@qnx.com

Abstract

Designers of embedded systems have become increasingly interested in the Linux operating system, largely because of its open source model. But, as it turns out, the standard Linux kernel can't deliver the hard realtime capabilities — such as predictable response times and microsecond latencies — that a large number of embedded systems demand. Several products have emerged to fill the Linux realtime gap, with mixed success. For instance, some vendors have taken a dual-kernel approach that provides a fragile runtime environment for realtime tasks, while forcing developers to write new drivers and system services — even when equivalent services already exist for Linux. Meanwhile, others have proposed a "pure" Linux solution that, to be effective, would require a rewrite of the Linux driver and virtual file system frameworks. In this paper, we look at an alternate approach — using a POSIX-based RTOS designed specifically for embedded systems — that not only allows Linux developers to keep their programming model, but also maintains the key advantages of Linux's open source model. As an added benefit, this approach allows embedded developers to enjoy OS services unavailable with either standard Linux or realtime Linux extensions. To illustrate the viability of this approach, the paper ends with examples of various Linux applications already ported QNX Neutrino, the POSIX-based RTOS from QNX Software Systems.

Filling the Realtime Gap

For the embedded systems designer, Linux poses a dilemma. On the one hand, Linux lets the designer leverage a large pool of developers, a rich legacy of source code, and industry-standard POSIX APIs. At the same time, the standard Linux kernel can't deliver the "hard" realtime capabilities — such as guaranteed response times and microsecond latencies — that many embedded devices require.

The reasons for this are rooted in Linux's general-purpose architecture. Take process scheduling, for instance. Rather than use a preemptive priority-based scheduler, as an RTOS would, Linux implements a "fairness" policy so that every process gets a reasonable opportunity to execute. As a result, high-priority, time-critical processes can't always gain immediate access to the CPU. In fact, the OS will sometimes interrupt a high-priority process to give a lower-priority process a share of CPU time.

Also, the standard Linux kernel isn't preemptible. A high-priority process can never preempt a kernel call, but must instead wait for the entire call to complete — even if the call was invoked by the lowest-priority process in the system. This makes it difficult, if not impossible, to create a design where critical events are consistently handled within short (and predictable) timeframes.

Mind you, it's wrong to think that this scheduling model constitutes a deficiency in Linux. The model does very well, for instance, at achieving the high overall throughput required by desktop and server applications. It only falls short when forced into deterministic environments that it wasn't designed for, such as network routers, factory robots, medical instruments, and continuous media applications.

Be that as it may, several products have emerged with the aim of filling in Linux's realtime gap. Some of these represent the work of commercial vendors. Others are open-source research projects. Some are a combination of both. No approach, however, has emerged as the "standard." In fact, some approaches even deviate from the standard Linux/POSIX programming model.

Real Time Implemented Outside of Linux

For instance, most vendors provide "realtime Linux" — note the quotation marks — by running Linux as a task on top of a realtime kernel (see Figure 1). Any tasks that require deterministic scheduling also run in this preemptible kernel, but at a higher priority than Linux. These tasks can thus preempt Linux whenever they need to execute and will yield the CPU to Linux only when their work is done.

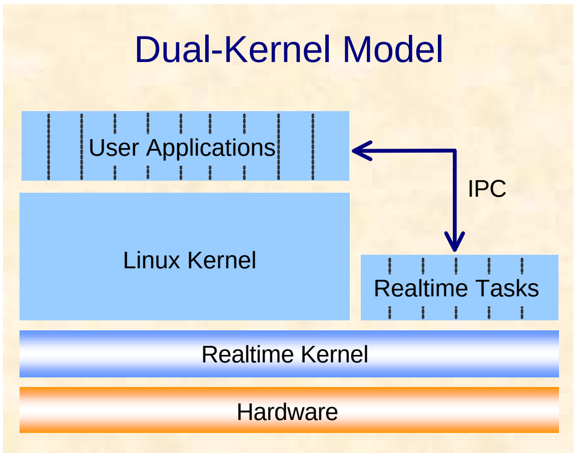

Figure 1 — In the dual-kernel model, Linux runs as the lowest-priority task in a separate realtime kernel.

In this dual-kernel model, the realtime kernel always gets first dibs on hardware interrupts. If an interrupt is flagged for a realtime task, the kernel will schedule that task to run. Otherwise, the kernel will pass the interrupt to Linux for processing. The nice thing here is that all this work is invisible to applications running in the Linux environment — except, of course, for the CPU cycles lost to the realtime kernel and its tasks. Still, the approach has several shortcomings:

Duplicated coding efforts

Tasks running in the realtime kernel can't make full use of existing Linux system services — file systems, networking, and so on. In fact, if a realtime task calls out to Linux for ***any*** service, it will be subject to the same preemption problems that prohibit Linux processes from behaving deterministically.

As a result, new drivers and system services must be created specifically for the realtime kernel — even when equivalent services already exist for Linux. Since few such drivers are available off the shelf, Linux developers must typically write them from scratch, using an unfamiliar API.

Fragile execution environment

Tasks running in the realtime kernel don't benefit from the robust MMU-protected environment that Linux provides for regular, non-realtime processes. Instead, they run unprotected in kernel space. Consequently, any realtime task that contains a common coding error, such as a corrupt C pointer, can easily cause a fatal kernel fault. That's a problem, since most systems that need real time also demand a very high degree of reliability.

Limited portability

With the dual-kernel approach, realtime tasks aren't Linux processes at all, but threads and signal handlers written to a small subset of the POSIX API or, in some cases, a non-standard API. Moving existing Linux code and applications to the realtime environment becomes difficult.

To complicate matters, different implementations of the dual-kernel approach use different APIs. Realtime tasks written for one vendor's realtime extensions may not run on another's. Embedded device manufacturers hoping to leverage Linux's widely supported APIs must instead choose between competing "standards."

No determinism for existing Linux applications and drivers

Because Linux processes don't run in the realtime kernel, they don't gain any deterministic behavior. Instead, they continue to be scheduled according to Linux's fairness algorithm.

Limited design options

As mentioned, the APIs supported by the realtime kernel provide only a subset of the services provided by standard POSIX and Linux APIs. Hence, developers have fewer design options than they would with either Linux or a mature RTOS.

"Pure" Realtime Linux?

Given the shortcomings of the dual-kernel approach, wouldn't it be better to make Linux itself realtime? Apparently, this can't be too hard to do, since at least one vendor has claimed they have a pure Linux kernel capable of supporting applications with "realtime requirements." In a nutshell, they made the kernel preemptible by enabling Linux's SMP locking mechanisms to work on a single processor.

This approach has merit, since it allows developers to use a standard Linux kernel and programming model. And, like other common approaches to making Linux more deterministic, such as adding high-resolution timers, it helps address the low-latency requirements of reactive, event-driven systems. Unfortunately, such low-latency patches don't address the complexity of most realtime environments, where realtime tasks span larger time intervals and have more dependencies on system services and other processes than do tasks in a simple event-driven system.

For instance, in systems where realtime tasks depend on services such as device drivers or file systems, the problem of priority inversion would have to be addressed within the Linux driver and virtual file system (VFS) models. This would effectively require a rewrite of those frameworks — and of all device drivers and file systems employing them. Without these modifications, realtime tasks could experience unpredictable delays when blocked on a service. As a further problem, most existing Linux drivers aren't preemptible. Thus, to ensure predictability, programmers would also have to insert preemption points into every driver in the system — something that few Linux developers have experience doing.

In any case, it's unclear which realtime modifications, if any, will eventually be integrated into the standard Linux kernel. After all, most Linux-based systems rely on the kernel's current approach, which sacrifices predictability to achieve higher overall throughput. A more deterministic kernel — with its attendant reduction in average response times — may simply be out of step with what most Linux developers want.

The Best of Both Worlds

The QNX® Neutrino® represents an entirely different and altogether more viable approach to the problems we've been discussing. Instead of trying to make developers squeeze predictable response times out of Linux, QNX Neutrino offers a proven realtime operating environment that:

- allows Linux developers to keep their existing APIs and programming model
- addresses the shortcomings of realtime Linux extensions through a much tougher runtime model, greater design options, and a unified environment for realtime and non-realtime applications
- maintains key benefits associated with Linux's open source model, such as easier troubleshooting and OS customization — in fact, QNX Neutrino's architecture offers distinct advantages on both counts

Compatible to the core

How well does QNX Neutrino support the Linux programming model? The answer lies, to a great degree, in the POSIX APIs adopted by Linux. Though rooted in Unix practice, these APIs were defined by the POSIX working groups purely in terms of "interface," not "implementation." Simply put, the APIs aren't tied to any OS or OS architecture. As a result, an RTOS can support the same POSIX APIs as Linux (along with various subtleties of Linux compatibility) without adopting Linux's non-deterministic kernel.

The QNX Neutrino RTOS supplies proof for this implementation-independent approach: It offers POSIX-compliant APIs, but implemented on a realtime, microkernel architecture (see Figure 2). Importantly, QNX doesn't implement POSIX as an add-on layer. Rather, the very core of the QNX Neutrino RTOS — the QNX microkernel — was designed from the beginning to support POSIX realtime facilities, including threads. POSIX, and hence Linux, compatibility runs deep.

Embedded Linux: a new definition

This issue of API compatibility is taking on surprising importance as OEMs attempt to use Linux in

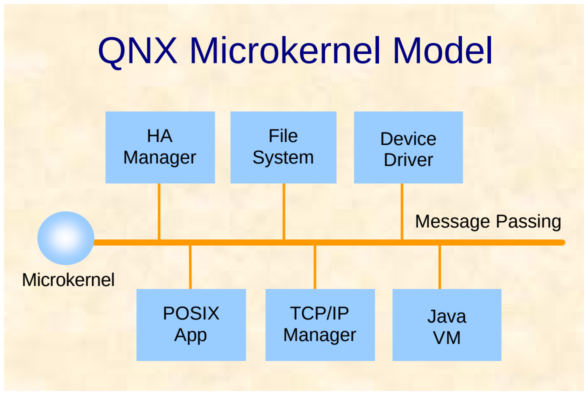

Figure 2 — In QNX Neutrino, the microkernel contains only the most fundamental OS services. All other services are provided through optional, memory-protected processes that can be stopped and started dynamically. To achieve this modularity, QNX Neutrino uses message passing as the fundamental means of IPC for the entire system.

their embedded products. Until recently Linux was, more than anything else, a unified community of developers creating software for a relatively narrow range of environments — mostly web servers and workstations based on x86 hardware. This common focus meant that Linux developers could count on a variety of benefits: binary compatibility, a consistent OS kernel, a pool of readily adaptable source code, and so on.

Unfortunately, no such common focus is possible in the embedded market, for the simple reason that embedded systems are so incredibly diverse. Most are purpose-specific, requiring custom applications, custom drivers, custom OS services, custom board designs, and so on. In fact, the consequences of adapting Linux to this diversity are fast becoming obvious. Scores of embedded OEMs and developers are now independently rolling their own Linux kernels with the result that, instead of one Linux, many non-standard and incompatible versions exist. Fragmentation isn't a distant threat; it's happening today.[1]

There is, then, a need for a new *lingua franca* among embedded Linux developers — a way to define "embedded Linux" that accommodates the teeming diversity of the embedded market, while providing a critical mass of portability and interoperability. That's exactly what the Embedded Linux Consortium (ELC), a vendor-neutral trade association dedicated to advancing Linux in embedded markets, is now doing with its soon-to-be-proposed ELC Platform Specification. This specification will define embedded Linux in a new way: as a set of APIs, along with a test suite to measure conformance.

Truth is, this approach isn't really new. The POSIX specification did much the same thing when it helped save the Unix world from fragmentation. In fact, the ELC specification will be based on existing POSIX standards, such as the POSIX 1003.1, which the QNX Neutrino RTOS supports today. As a result, the QNX Neutrino RTOS will inherently support embedded Linux applications, while simultaneously providing all of the benefits of a true RTOS designed from the ground up for embedded systems (more on these benefits later).

Inherently Open

Still, Linux compatibility is a red herring if an RTOS doesn't also address why most developers consider Linux in the first place: the benefits of its open source model. With open source, developers can analyze the architecture of the OS to better integrate their own code, adapt OS components to application-specific demands, and save considerable time troubleshooting — not only when problems occur in their own programs, but when a problem involves unexpected results from underlying OS code. In short, developers gain a level of vendor independence and self-sufficiency not possible with the "black box" model of many commercial OSs.

The QNX Neutrino RTOS offers these benefits in two ways: 1) by using a highly extensible microkernel architecture; and 2) by providing customers with source code for drivers, libraries, and BSPs, including well-documented driver development kits for a variety of standard devices.

As a microkernel OS, QNX Neutrino is fundamentally open to customization. Except for a few core services (e.g. scheduling, timers, interrupt handling) that reside in kernel space, most OS-level services — drivers, file systems, protocol stacks, and so on — exist as user-space applications outside the kernel. As a result, developing custom drivers and application-specific OS extensions doesn't require specialized kernel debuggers or kernel gurus. In fact, as user-space programs, OS extensions become as easy to develop as standard applications, since they can be

[1] Fragmentation raises another issue: lower reliability. A key reason why standard x86 Linux performs so reliably is that a large community of developers is continually locating and fixing problems in the code. That benefit no longer applies when you deploy a custom Linux kernel supported by only a few developers.

debugged with standard, source-level tools familiar to every Linux developer.

Better yet, QNX Neutrino allows applications to access all drivers and OS services via a single, consistent form of IPC: synchronous message passing. This approach offers several advantages. For instance, because QNX message passing is synchronous, it automatically coordinates the execution of communicating programs, thereby eliminating the need to handcode — and debug — complex synchronization services in every process (see Figure 2). Moreover, message passing inherently simplifies the task of partitioning a complex system into well-defined building blocks that can be developed, tested, and maintained individually. Troubleshooting becomes much easier as a result.

Does this mean, however, that developers must use proprietary message-passing functions? Not at all. QNX messaging is implemented so seamlessly that applications and OS services don't have to invoke special functions to exchange messages — the messages can be sent and received transparently, using standard POSIX calls. Now, that may sound a bit like magic, so let's look at how it works.

In QNX Neutrino, any service-providing program (e.g. a driver) can "advertise" its services to other programs by registering a pathname in the pathname space. Programs can then access those services by

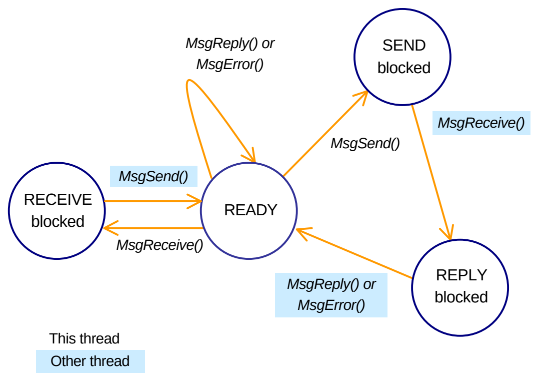

Figure 3 — QNX message passing simplifies the synchronization of processes and threads. For instance, the act of sending a message automatically causes the sending thread to be blocked and the receiving thread to be scheduled for execution.

issuing calls such as *open*(), *read*(), *write*(), or *lseek*() on the pathname. For instance, the QNX serial-port driver typically registers the pathname /dev/ser1 to represent the first serial port. Any application that needs to access that port simply issues an *open*() on /dev/ser1.

So where does the message passing come in? Well, from the application's perspective, the *open*() looks and behaves like a standard POSIX call; there's nothing special about it. But, underneath the hood, the QNX Neutrino C library converts the call into an io_open message and forwards the message to the serial driver. If the application subsequently wished to write a character to the serial port, a similar sequence would occur: the client would issue a *write*(), the C library would construct an io_write message, and the message would be sent to the driver.

There's an interesting and highly beneficial side-effect here: the exact same message can be sent from any client application to any service, provided the service supports that particular function. For instance, to write a character to a serial port or to put a character into a disk file, an application would issue the same *write*() function in either case; the only difference would be where the message was sent. In other words, the application is cleanly decoupled from the services it relies on. This decoupling simplifies development since all interactions between applications and system services can be implemented using a simple, POSIX-based programming model. It also eases migration to new designs since applications don't have to include hardware- or protocol-specific code.

Two more (short) points about QNX message passing before we move on. First, developers can access the QNX messaging framework directly, using three simple calls: *MsgSend*(), *MsgReceive*(), and *MsgReply*(). Second, message passing isn't the only form of IPC that QNX Neutrino supports. Developers can, in fact, work with several standard forms of IPC, including signals, POSIX message queues, shared memory, pipes, and FIFOs.

Available source

QNX Neutrino further simplifies troubleshooting and OS customization by providing customers with full source code for libraries, drivers, and board-support packages. Developers can also freely download device driver kits (DDKs) for a variety of device types, including networking, graphics, input, audio, and USB. Besides providing source code, the DDKs include clear, concise documentation and a software framework that implements all higher-level, device-independent code in libraries. The only code the developer has to write is the hardware-specific code for the chip on their device.

While discussion of source licensing models is beyond the scope of this article, it's important to note that this QNX source code isn't provided under the GPL that covers most Linux source. Rather, QNX Software Systems provides the source under its own license agreement, which — unlike the GPL — gives developers the freedom to create derivative works without having to sacrifice any of their own intellectual property (IP). Put simply, QNX source is free of the IP issues that prevent many embedded system manufacturers from using GPL-covered code.

Real(time) Benefits

Implementing POSIX/Linux APIs on a microkernel architecture also addresses the drawbacks associated with the realtime extensions discussed earlier:

A tougher runtime model

As a monolithic OS, Linux binds most drivers, file systems, and protocol stacks to the OS kernel. Hence a single programming error in any of these components can cause a fatal kernel fault. In QNX Neutrino, these components can all run in separate, memory-protected address spaces, so it's very difficult for them to corrupt the kernel, or each other. QNX Neutrino therefore provides an environment for realtime applications that is inherently more robust than Linux — and certainly much tougher than the unprotected realtime kernels used in the dual-kernel approach.

A unified environment

In QNX, the realtime and non-realtime environments are one and the same. Realtime applications can take advantage of the full POSIX API and enjoy full access to system services — GUIs, file systems, and so on. By the same token, existing POSIX/Linux applications can immediately gain deterministic behavior. And since both realtime and non-realtime applications are running in the same message-based environment, IPC between them is greatly simplified.

Less duplicated effort

As discussed, the dual-kernel approach can force developers to write custom drivers, using an unfamiliar API. As in most OS environments, developing these drivers requires kernel debugging tools (hard to use), kernel rebuilds (time-consuming), and kernel programmers (expensive).

QNX Neutrino addresses this problem in several ways. First, like any established OS with a large user base, QNX supports a variety of off-the-shelf drivers for standard hardware. And, as we've seen, QNX Neutrino runs all drivers in user space, so they can be developed using standard source-level tools and techniques. This job is made all the easier by the QNX DDKs, which provide documentation, libraries, headers, and ready-to-customize source for a variety of drivers.

Additional Microkernel Features

As a microkernel RTOS designed specifically for the demands of embedded systems, QNX Neutrino also offers Linux developers features unavailable with either standard Linux or realtime Linux extensions. These include:

Built-in distributed processing

QNX Neutrino provides an OS service, the *QNX micronetwork*, that allows messages to flow transparently across processor boundaries. Consequently, any process can, given appropriate permissions, access virtually any resource on any other node, as if that resource were local.

For instance, if an application wishes to send an open message to a device driver, it doesn't matter whether the driver is local or remote: the application sends the exact same *open*() call in either case. If the pathname for the driver is local, the QNX microkernel will route the message directly; if the driver is on a remote node, the QNX micronetwork will transparently forward the message to that node.

Fault-tolerant networking

Thanks to the network abstraction provided by QNX message passing, applications can communicate transparently over redundant network links: if one link fails, the OS will automatically reroute traffic over the remaining links. Network traffic can also be load-balanced over all available links, resulting in higher throughput. Again, this service is built-in; applications require no special networking code.

Smaller memory footprint

Because of the fine-grained scalability of microkernel architecture, the QNX RTOS can provide a runtime environment considerably smaller than Linux — a critical advantage in high-volume devices such as information appliances and in-car telematics systems, where even a $2 reduction in memory costs per unit can return millions of dollars in profits. In fact, QNX's native windowing system, the Photon microGUI®, also uses a microkernel architecture, so designers can easily "unplug" GUI services that aren't required by their memory-constrained devices.

Consistent, field-tested kernel

Unlike the monolithic Linux kernel, which can change from embedded system to embedded system, the QNX microkernel can be used unmodified across an incredibly diverse range of products. In fact, there is only one QNX microkernel binary for each family of supported CPUs. Developers have the assurance that they're using the same microkernel lab-tested at QNX Software Systems and field-tested in other customer installations.

A Matter of Synergy

While QNX Neutrino offers a superior platform for running realtime applications, choosing between it and Linux doesn't have to be a mutually exclusive, either/or proposition. In fact, because the two OSs share so much common ground, developers can readily target both OSs, using each where it fits best. Moreover, QNX supports TCP/IP, NFS, and even a Linux file system, so it's easy for a development shop that uses a mix of Linux and QNX workstations to share resources across the two environments.

In short, Linux and QNX Neutrino do more than simply coexist. Rather, they give developers the opportunity to leverage many of the same APIs, source code, and skill sets across a much wider spectrum of applications than any operating system — whether realtime or general-purpose — could do on its own.

Supplement: Porting Open Source Applications to QNX Neutrino

Since QNX Neutrino is a POSIX operating system, you can port most Linux applications and other open source programs with little effort. In most cases, you simply recompile the source and relink with QNX Neutrino libraries. System administration, networking, database, and computational programs are all examples of applications that can be ported this way.

The following table lists a sample of the open source applications that have already been ported to the QNX Neutrino RTOS. For a more comprehensive list, visit http://www.qnx.com/developer/download.

GCC*	GNU C/C++ compiler
CVS*	Source code versioning control system
Vim	Vi IMproved, a programmers' editor; the GUI version (gvim) has been ported to QNX Photon microGUI
GNU EMACS	Programmers' editor
GDB*	GNU debugger
DDD	Graphical debugger that uses GDB as its back end for debugging (the GUI is X)
Doxygen	Source code documentation tool
Perl	Scripting language
Python	Scripting language
Ruby	Scripting language
Apache	Web server
Mozilla*	Web browser based on the Netscape source code
Pine	Email client
Mutt	Email client
Sendmail	Email server
Tin	Newsgroup reader
Samba	Provides seamless access to files, printers, and other shared resources on Windows networks
Open SSH/SSL	Secure sockets and shells
Open LDAP	Light weight Directory Access Protocol
PVM	Distributed processing system
Zebra router	Software for managing TCP/IP-based routing protocols
The GIMP	GNU Image Manipulation Program; similar to Photoshop, uses X as its GUI
Abiword	Word processor
Quake III	Multimedia game

* Only available as part of the free QNX Momentics NC edition, which can be downloaded from http://www.qnx.com/nc.