A
pplication migration from one Operating System (OS) to another is often a daunting task, even in the best of cases. Migrating a real-time embedded application to a new OS is arguably among the most difficult tasks. To aid developers planning to move to embedded Linux in the near future or considering the level of investment necessary to convert their existing applications to run on embedded Linux, Jim explains the transition process, assesses the challenges involved, and presents the benefits realized from such a move.

More and more companies that previously used a Real-Time Operating System (RTOS) as their embedded OS are turning to embedded Linux as the OS for their next generation of products. In fact, industry analyst Venture Development Corporation shows embedded Linux and open source garnering up to one-third of 32- and 64-bit designs, more than twice the share of any other embedded OS.[1]

Clearly, questions about the feasibility of application migration from older RTOS-based products to Linux must be answered so this migration can be managed efficiently.

Characterizing the challenge
A typical RTOS-based application is dependent upon many factors. Among the most significant are the programming/memory model, API set and performance, and especially real-time responsiveness. Another serious consideration is the software development environment, but that is a discussion worthy of its own article.

Programming model
Virtually all widely used RTOSs have a simple programming model consisting of multiple threads of execution (often called tasks, as in multitasking) all contained in a single address space. For example, a C language program has a single main function from which all other threads are created. Each thread in turn is defined as a C function within the overall program. Typically both the RTOS and the application reside in unprotected memory, where physical and logical addresses are the same. There may be some use of supervisor/user modes of operation that restrict applications in user mode from issuing certain instructions, thereby adding modest protection. Fundamentally, though, all memory is visible to all the application, which has both pluses (performance and simplicity) and a big minus (vulnerability to memory corruption bugs).

In the past, most embedded microprocessors did not have a Memory Management Unit (MMU), so the single address space RTOS model was a necessity. However, most mid- and high-end microprocessors today are equipped with MMUs, so the MMU can manage memory if desirable.

This architecture description suggests a straightforward architecture for porting RTOS code to Linux:

- The entirety of RTOS application code (minus kernel and libraries) migrates into a single Linux process.
- RTOS tasks translate to Linux threads.
- RTOS physical memory spaces (that is, entire system memory complements) map into Linux virtual address spaces.
A multiboard or multiprocessor architecture such as a VME rack migrates into a multiprocess Linux application, as in Figure 1.

Architectural considerations: process and thread creation

Whether using RTOS emulation kits for Wind River VxWorks and pSOS (a discussion of which follows in the API section) or performing the port unaided, developers ultimately must decide whether to implement RTOS tasks as processes or threads. At its heart, the Linux kernel treats both processes and threads as coequal for scheduling purposes. However, different APIs create and manage each type of entity, and performance, resource costs, and benefits are associated with each.

In general, processes are heavier than threads because they carry more context. A Linux thread context such as an RTOS task consists primarily of a subset of CPU registers, stack, current program counter, and some entries in the kernel’s data structures (task control blocks in an RTOS). A process adds a complete virtual address space. Thus, at a minimum, the kernel must also create and track page translations and types for all code, constant text, and data used by the process. The major impact of this heavier process context comes at two junctures: process creation time and interprocess context switch time.

RTOS code strives for lightweight execution whenever possible. As such, while many RTOSs offer dynamic task creation APIs, others feature only static task definition tables, and all RTOS vendors discourage frivolous and frequent task creation to save time and space. Linux is prevalent in UNIX-based open systems, and many customers also layered applications through the brute-force use of watchdog timers that reboot the entire system or software-induced panics.

Most often when a program goes awry, it does so silently. An errant task can corrupt data and code anywhere in the RTOS system. With luck, the impact of such corruptions arises immediately (illegal instructions generating exceptions), but it is more likely that the damage will only surface after the fact—seconds, hours, or months later. When aberrant symptoms appear, it will be extremely difficult to associate unexpected program behavior, whether subtle or crash and burn, with the original cause.

This timeworn and familiar architecture, while simple, is highly exposed to corruption. Runaway tasks can overwrite application code and data, accidentally write into peripheral device registers, corrupt kernel data structures, and overwrite the kernel code. Tightly packed task stacks can easily underflow and overwrite one another or charge down through memory to corrupt the top of the heap, other data, or code laid out nearby.

At a higher level, this informally organized and highly exposed architecture presents two key challenges to code quality: the scope of the failure itself and the association of second-order failures with the primary event.

When an individual task or other software component fails, the scope of its failure is almost impossible to determine, let alone realize that it failed at all. Even when a failure is detected and recovery attempted, the granularity of failure ends up being the entire system. Monitor code cannot usually restart tasks safely, and the RTOS cannot recover resources dynamically allocated by failed tasks. The result is that recovery is most often accomplished through the brute-force use of watchdog timers that reboot the entire system or software-induced panics.

Built-in reliability from the Linux programming model

Linux, as a UNIX-compatible OS, presents a much more robust application and system programming model. Applications execute in their own protected address spaces, for the most part invisible to one another, and are prevented from overwriting each other’s data or code.

While they share this virtual address space with the Linux kernel, they cannot overwrite kernel code or data. Since applications/processes cannot “see” one another (they reside in unique virtual address spaces), they cannot corrupt each other’s data or code.

API (native and standard) and runtime libraries

Before open standards, RTOS makers defined their own system call set or API, which was proprietary and unique to each maker. Interface libraries were supplied for popular programming languages such as C and C++, which made APIs accessible to the programmer writing in a high-level language. In the past decade, most RTOS makers also supplied libraries compliant with the well-established standard Portable OS Interface (POSIX), although typically only the portion of the POSIX specifications relevant to embedded applications. Many customers also layered the native RTOS interface with their own API set to gain some independence and portability, not wanting to be locked into a proprietary, vendor-specific interface.

Developers build applications that leverage standard APIs to achieve two complementary purposes: allow code to be ported to standards-based OSs like Linux and allow that same code later to be ported from such an environment more easily than with proprietary APIs.

Many commercial RTOSs include standard call sets from POSIX or Berkeley Software Distribution (BSD), but those APIs often exist only as window dressing. The proprietary closed APIs particular to a given kernel are the most used, and it is these that lock projects to a particular platform or solution.

If developers are porting standards-based code or considering which API options to choose for new code, it is important to understand the most common standards in use under Linux and other open systems.

POSIX

POSIX is prevalent in UNIX-based open systems and government and military
arenas. However, POSIX has had limited impact on the traditionally closed and proprietary world of embedded RTOSs. The POSIX family of standards originated by the U.S.-based National Institute of Standards and Technology now falls under the auspices of IEEE as IEEE 1003 and other standards. In the past decade, POSIX has undergone several revisions, most recently to POSIX 2000.

Two important notions associated with POSIX are compliance and conformance. Compliance implies that a given OS platform implements some subset of the standard and that the implementation is documented. Even those platforms implementing a trivial subset can be termed POSIX compliant. POSIX conformance, conversely, presents much stricter criteria, meaning that an OS subjected to a certification test passed.

**SVR4, BSD, and other UNIX APIs**

UNIX System V, Revision 4 (SVR4) and versions of BSD UNIX are prevalent de facto system standards that greatly influence Linux. Linux implements large subsets of those UNIX APIs (for example, the Linux ip() system call for shared memory, queues, and semaphores and the BSD sockets calls and TCP/IP stack).

Developers familiar with SVR4, BSD, or other common UNIX implementations like AIX, HP-UX, and so on will feel right at home on Linux.

**C language libraries**

Many of the APIs in embedded designs, RTOS-based and otherwise, are simply standard C libraries that either directly implement functions or act as wrappers for system calls. Linux has the familiar libc/glibc, although larger in scope and more comprehensive than in RTOS implementations (for instance, pREPC for pSOS).

The glibc runtime can present memory footprint challenges to embedded applications, migrated and otherwise. Many Linux suppliers offer reduced code-sized libraries for size-sensitive applications.

**RTOS interface layers**

At the heart of an RTOS-based application is the use of InterProcess Communication (IPC) calls that supply the mechanisms to synchronize and communicate among tasks. Although the acronym uses the word process, in an RTOS, it really refers to tasks.

Table 1 provides a summary of the mapping between typical RTOS IPC calls and their equivalent Linux calls.

Although the mapping between an RTOS call and its equivalent Linux call is straightforward, the migration effort may be further accelerated using RTOS emulation libraries that supply the identical call interface for a Linux application as found on the RTOS being migrated from. One such emulation technology is available at the Xenomai Open Source project (www.xenomai.org). Here, various emulation layers, called skins in the Xenomai parlance, are supplied for POSIX, VxWorks VRTX, and Itron, which are widely used RTOSs. Note that like many open source projects, Xenomai and its skins are a work in progress and may be incomplete and/or subject to change. Nevertheless, it represents a starting point of potentially high value in streamlining the migration process.

For example, the POSIX module on top of Xenomai aims at providing an (almost) PSE51-compatible API. To help porting applications from other PSE51-compatible APIs, it contains some nonportable extensions to the POSIX specification.

This POSIX skin already contains these basic features:

- **Threads**
- **Mutexes**
- **Semaphores (anonymous and named)**
- **Condition variables**
- **Support for real-time signals**
- **Cancellation, cancellation handlers**
- **Thread-specific data**
- **Message queues**
- **Timers support**
- **Shared memory**

The POSIX skin creates real-time threads running either embodied into Linux kernel modules or inside regular applications in user space.

The APIs of real-time kernels allow both kernel and user space programming. When using these, developers generally prefer user space programming because the latency gap between both spaces is small, especially on hardware where MMU context switching is cheap. The ease of user space programming outweighs by far any gain one could expect from running an application directly from kernel space. User space programming brings memory protection and GNU Debugger support to debug real-time applications in this environment.

**Real-time performance**

Perhaps the most important requirement for embedded applications is meeting real-time deadlines. Considerable effort has gone into both designing RTOSs to make them responsive enough to meet real-time deadlines and into measuring RTOS system calls, so developers can be sure the systems perform quickly enough to meet deadlines. The RTOS calls are in the loop in the sense that an application synchronizes with an interrupt using services supplied by the RTOS. Consequently, the time it takes the RTOS to process a synchronization call is part of the timing-dependent sequence of handling the interrupt.

Before 2002, Linux’s real-time performance was poor. Its throughput, especially in networking, was good; however, that’s throughput, not real time. The reason was the basic Linux kernel and UNIX implementation architecture. These systems were designed for the kernel to execute what it needed to at the expense of the application. The reason was that kernel code is easier to write if developers know it won’t be preempted by an execution path interruption.

---

**Table 1**

<table>
<thead>
<tr>
<th>RTOS IPCs</th>
<th>Linux IPCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphores (counting and binary)</td>
<td>SVR4 semaphores</td>
</tr>
<tr>
<td>Mutexes</td>
<td>POSIX.1c mutexes, condition variables, futexes</td>
</tr>
<tr>
<td>Message queues and mailboxes</td>
<td>Pipes/FIFOs, SVR4 queues</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Shared memory</td>
</tr>
<tr>
<td>Events and RTOS signals</td>
<td>Signals, RT signals</td>
</tr>
<tr>
<td>Timers, task delay</td>
<td>POSIX timers/alarms sleep() and nanosleep()</td>
</tr>
<tr>
<td>Watchdogs, task regs, partitions/buffers</td>
<td>Emulated by tool kits</td>
</tr>
</tbody>
</table>
This approach served UNIX and early Linux to a degree, but there was a downside: It made running on multiple CPUs inefficient. At the same time, nonpreemptible Linux made meeting real-time requirements difficult because even though an interrupt may have occurred and an application was scheduled to run, the kernel would finish whatever work it was doing at the expense of that application. To fix the problem of running efficiently on multiprocessor systems, Linux kernel developers started breaking the internal paths of the Linux kernel into smaller non-preemptible paths so that more of the kernel could execute in parallel and perform better on multiprocessor machines.

Those interested in improving the real-time performance of Linux piggybacked on the multiprocessing work and used the now-preemptible Linux kernel to accelerate native real-time response. That step alone reduced delays from many hundreds of milliseconds down to the 1 millisecond range. Further improvements included shortening the nonpreemptible sections to reduce any delay these smaller sections might add.

Linux has supported real-time applications since 2002, when Linux developers began enhancing its real-time capabilities. Since then, a continuous improvement process has dramatically accelerated the real-time response capabilities of standard Linux. Current real-time capabilities of Linux are equivalent to that of most dedicated real-time kernels.

A significant boost in moving Linux to real-time capability occurred recently when spin locks, which are synchronization mechanisms that consume CPU cycles but protect nonpreemptible sections, were replaced by a more reliable synchronization mechanism called priority inheritance mutexes. Mutexes make sure that CPU time is always allocated as closely as possible to the highest-priority application. This further shortens the processing path from interrupt to real-time application.

The last major real-time Linux improvement was to elevate the processing of the bulk of interrupt handling to the application level. Previously, the Linux design had all the interrupt handling at a higher priority than any application. By moving the bulk of the interrupt handling to application level, once again priority could take hold and the highest-priority application could get time before a lower-priority interrupt handler.

Now that these changes have been completed, the performance and reliability improvements have made native Linux applications as fast and reliable as those based on traditional RTOSs.

Moving forward
Developers are leaving first-generation RTOSs for more reliable and open embedded platforms like Linux. While the migration from these traditional systems presents challenges, the benefits far outweigh the investment needed. The real risk doesn’t arise from leaving behind familiar environments, tools, and APIs; it lies in standing still while the embedded systems development community moves forward.

By following the steps outlined in this article and leveraging RTOS migration technology, developers can successfully migrate existing legacy RTOS code to a modern embedded Linux platform with minimal effort and time.

Jim Ready is the founder and CTO of MontaVista Software, Inc. in Santa Clara, California. With 25-plus years of technical and entrepreneurial experience, Jim is a recognized authority in the embedded systems and real-time software industries. Cofounder of Ready Systems, he pioneered the development of the first commercially viable RTOS product, the VRTX real-time kernel. During the period when Ready Systems merged with Microtec Research, which later was acquired by Mentor Graphics, Jim served as Ready Systems’ president and CTO at Microtec/Mentor Graphics. He founded MontaVista Software in 1999 to offer the Linux OS to the embedded systems market and provide embedded system expertise to the open source Linux community. Jim has BA and MA degrees in Physiological Psychology from the University of Illinois at Urbana-Champaign and the University of California, Berkeley, respectively.

MontaVista Software, Inc.        
408-572-8000        
jready@mvista.com       
www.mvista.com

References