Embedded

Porting RTOS Device Drivers to Embedded Linux

Transform your wild-and-woolly legacy RTOS code into well-formed Linux device drivers.

by Bill Weinberg

Linux has taken the embedded marketplace by storm. According to industry analysts, one-third to one-half of new embedded 32- and 64-bit designs employ Linux. Embedded Linux already dominates multiple application spaces, including SOHO networking and imaging/multifunction peripherals, and it now is making vast strides in storage (NAS/SAN), digital home entertainment (HDTV/PVR/DVR/STB) and handheld/wireless, especially in digital mobile phones.

New embedded Linux applications do not spring, Minerva-like, from the heads of developers; a majority of projects must accommodate thousands, even millions of lines of legacy source code. Although hundreds of embedded projects have successfully ported existing code from such platforms as Wind River's VxWorks and pSOS, VRTX, Nucleus and other RTOSes to Linux, the exercise is still nontrivial.

To date, the majority of literature on migration from legacy RTOS applications to embedded Linux has focused on RTOS APIs, tasking and scheduling models and how they map to Linux user-space equivalents. Equally important in the I/O-intensive sphere of embedded programming is porting RTOS application hardware interface code to the more formal Linux device driver model.

This article surveys several common approaches to memory-mapped I/O frequently found in legacy embedded applications. These range from ad hoc use of interrupt service routines (ISRs) and user-thread hardware access to the semi-formal driver models found in some RTOS repertoires. It also presents heuristics and methodologies for transforming RTOS code into well-formed Linux device drivers. In particular, the article focuses on memory mapping in RTOS code vs. Linux, porting queue-based I/O schemes and redefining RTOS I/O for native Linux drivers and daemons.

RTOS I/O Concepts

The word that best describes most I/O in RTOS-based systems is informal. Most RTOSes were designed for older MMU-less CPUs, so they ignore memory management even when an MMU is present and make no distinction between logical and physical addressing. Most RTOSes also execute entirely in privileged state (system mode), ostensibly to enhance performance. As such, all RTOS application and system code has access to the entire machine address space, memory-mapped devices and I/O instructions. Indeed, it is very difficult to distinguish RTOS application code from driver code even when such distinctions exist.

This informal architecture leads to ad hoc implementations of I/O and, in many cases, the complete absence of a recognizable device driver model. In light of this egalitarian non-partitioning of work, it is instructive to review a few key concepts and practices as they apply to RTOS-based software.

In-Line Memory-Mapped Access
When commercial RTOS products became available in the mid-1980s, most embedded software consisted of big mainline loops with polled I/O and ISRs for time-critical operations. Developers designed RTOSes and executives into their projects mostly to enhance concurrency and aid in synchronization of multitasking, but they eschewed any other constructs that got in the way. As such, even when an RTOS offered I/O formalisms, embedded programmers continued to perform I/O in-line:

```c
#define DATA_REGISTER 0xF00000F5

char getchar(void) {
    return *((char *) DATA_REGISTER);
}

void putchar(char c) {
    *((char *) DATA_REGISTER) = c;
}
```

More disciplined developers usually segregate all such in-line I/O code from hardware-independent code, but I have encountered plenty of I/O spaghetti as well. When faced with pervasive in-line memory-mapped I/O usage, embedded developers who are new to Linux always face the temptation to port all such code as-is to user space, converting the #define of register addresses to calls to mmap(). This approach works fine for some types of prototyping, but it cannot support interrupt processing, has limited real-time responsiveness, is not particularly secure and is not suitable for commercial deployment.

### RTOS ISRs

In Linux, interrupt service is exclusively the domain of the kernel. With an RTOS, ISR code is free-form and often indistinguishable from application code, other than in the return sequence. Many RTOSes offer a system call or macro that lets code detect its own context, such as the Wind River VxWorks intContext(). Common also is the use of standard libraries by ISRs, with accompanying reentrancy and portability challenges.

Most RTOSes support the registration of ISR code and handle interrupt arbitration and ISR dispatch. Some primitive embedded executives, however, support only direct insertion of ISR start addresses into hardware vector tables. Even if you attempt to perform read and write operations in-line in user space, you have to put your Linux ISR into kernel space.

### RTOS I/O Subsystems

Most RTOSes ship with a customized standard C run-time library, such as pREPC for pSOS, and selectively patched C libraries (libc) from compiler ISVs. They do the same for glibc. Thus, at a minimum, most RTOSes support a subset of standard C-style I/O, including the system calls open, close, read, write and ioctl. In most cases, these calls and their derivatives resolve to a thin wrapper around I/O primitives. Interestingly, because most RTOSes did not support filesystems, those platforms that do offer file abstractions for Flash or rotating media often use completely different code and/or different APIs, such as pHILE for pSOS. Wind River VxWorks goes further than most RTOS platforms in offering a feature-rich I/O subsystem, principally to overcome hurdles in integration and generalization of networking interfaces/media.

Many RTOSes also support a bottom-half mechanism, that is, some means of deferring I/O processing to an interruptible and/or preemptible context. Others do not but may instead support mechanisms such as interrupt nesting to achieve comparable ends.
Typical RTOS Application I/O Architecture

A typical I/O scheme (input only) and the data delivery path to the main application is diagramed in Figure 1. Processing proceeds as follows:

- A hardware interrupt triggers execution of an ISR.
- The ISR does basic processing and either completes the input operation locally or lets the RTOS schedule deferred handling. In some cases, deferred processing is handled by what Linux would call a user thread, herein an ordinary RTOS task.
- Whenever and wherever the data ultimately is acquired (ISR or deferred context), ready data is put into a queue. Yes, RTOS ISRs can access application queue APIs and other IPCs—see the API table.
- One or more application tasks then read messages from the queue to consume the delivered data.

Output often is accomplished with comparable mechanisms—instead of using write() or comparable system calls, one or more RTOS application tasks put ready data into a queue. The queue then is drained by an I/O routine or ISR that responds to a ready-to-send interrupt, a system timer or another application task that waits pending on queue contents. It then performs I/O directly, either polled or by DMA.

Mapping RTOS I/O to Linux

The queue-based producer/consumer I/O model described above is one of many ad hoc approaches employed in legacy designs. Let us continue to use this straightforward example to discuss several possible (re)implementations under embedded Linux.

Developers who are reticent to learn the particulars of Linux driver design, or who are in a great hurry, likely try to port most of a queue-based design intact to a user-space paradigm. In this driver-mapping scheme, memory-mapped physical I/O occurs in user context by way of a pointer supplied by mmap():

Figure 1. Comparison between Typical I/O and Data Delivery in a Legacy RTOS and Linux
#include <sys/mman.h>

#define REG_SIZE   0x4   /* device register size */
#define REG_OFFSET 0xFA400000
    /* physical address of device */

void *mem_ptr;
    /* de-reference for memory-mapped access */
int fd;

fd=open("/dev/mem",O_RDWR);
    /* open physical memory (must be root) */
mem_ptr = mmap((void *)0x0,
             REG_AREA_SIZE, PROT_READ+PROT_WRITE,
             MAP_SHARED, fd, REG_OFFSET);
    /* actual call to mmap() */

A process-based user thread performs the same processing as the RTOS-based ISR or deferred task would. It then uses the SVR4 IPC msgsnd() call to queue a message for receipt by another local thread or by another process by invoking msgrcv().

Although this quick-and-dirty approach is good for prototyping, it presents significant challenges for building deployable code. Foremost is the need to field interrupts in user space. Projects such as DOSEMU offer signal-based interrupt I/O with SIG (the silly interrupt generator), but user-space interrupt processing is quite slow—millisecond latencies instead of tens of microseconds for a kernel-based ISR. Furthermore, user-context scheduling, even with the preemptible Linux kernel and real-time policies in place, cannot guarantee 100% timely execution of user-space I/O threads.

It is highly preferable to bite the bullet and write at least a simple Linux driver to handle interrupt processing at kernel level. A basic character or block driver can field application interrupt data directly in the top half or defer processing to a tasklet, a kernel thread or to the newer work-queue bottom-half mechanism available in the 2.6 kernel. One or more application threads/processes can open the device and then perform synchronous reads, just as the RTOS application made synchronous queue receive calls. This approach will require at least recoding consumer thread I/O to use device reads instead of queue receive operations.

To reduce the impact of porting to embedded Linux, you also could leave a queue-based scheme in place and add an additional thread or daemon process that waits for I/O on the newly minted device. When data is ready, that thread/daemon wakes up and queues the received data for use by the consuming application threads or processes.

Porting Approaches

Porting RTOS code to embedded Linux does not differ conceptually from enterprise application migration. After the logistics of porting have been addressed (make/build scripts and methods, compiler compatibility, location of include files and so on), code-level porting challenges turn on the issues of application architecture and API usage.

For the purposes of the discussion at hand, let us assume that the application part (everything except I/O-specific code) migrates from the RTOS-based system into a single Linux process. RTOS tasks map to Linux threads and intertask IPCs map to Linux inter-process and inter-thread equivalents.
Although the basic shape of the port is easy to understand, the devil is in the details. And the most salient details are the RTOS APIs in use and how to accommodate them with Linux constructs.

If your project is not time-constrained, and if your goal is to produce portable code for future project iterations, then you want to spend some time analyzing the current structure of your RTOS application and how/if it fits into the Linux paradigm. For RTOS application code, you want to consider the viability of one-to-one mapping of RTOS tasks onto Linux process-based threads and whether to repartition the RTOS application into multiple Linux processes. Depending on that decision, you should review the RTOS IPCs in use to determine proper intra-process vs. inter-process scope.

On the driver level, you definitely want to convert any informal in-line RTOS code to proper drivers. If your legacy application already is well partitioned, either using RTOS I/O APIs or at least segregated into a distinct layer, your task becomes much easier. If ad hoc I/O code is sprinkled liberally throughout your legacy code base, you've got your work cut out for you.

Developers in a hurry to move off a legacy RTOS or those trying to glue together a prototype are more likely to attempt to map or convert as many RTOS APIs to Linux equivalents in situ. Entities in common, such as comparable APIs, IPCs and system data types, port nearly transparently. Others can be addressed with #define redefinition and macros. Those remaining need to be recoded, ideally as part of an abstraction layer.

You can get a head start on API-based porting by using emulation libraries that accompany many embedded Linux distributions (including MontaVista's libraries for Wind River VxWorks and pSOS) or by using third-party API-mapping packages from companies such as MapuSoft.
Most projects take a hybrid approach, mapping all comparable or easily translatable APIs, re-architecting where it doesn't slow things down and playing Whack-a-Mole with the remaining code until it builds and runs.

**Available APIs in Kernel and User Space**

For both intensive re-architecting and for quicker-and-dirtier API approaches, you still have to (re)partition your RTOS application and I/O code to fit the Linux kernel and user-space paradigm. Table 1 illustrates how Linux is much stricter about privileged operations than a legacy RTOS and helps guide you in the (re)partitioning process.

**Table 1. Privileged Operations in Linux and Legacy RTOSes**

<table>
<thead>
<tr>
<th></th>
<th>IPCs</th>
<th>Synchronization</th>
<th>Tasking</th>
<th>Namespace</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RTOS Application</strong></td>
<td>Queues, Signals, Mailboxes, Informal Shared Memory</td>
<td>Semaphores, Mutexes</td>
<td>Full RTOS Tasking Repertoire</td>
<td>Full Application, Libraries and System (Link-Time)</td>
</tr>
<tr>
<td><strong>RTOS Driver</strong></td>
<td>Queues, Signals, Mailboxes, Informal Shared Memory</td>
<td>Semaphores, Mutexes</td>
<td>Full RTOS Tasking Repertoire</td>
<td>Full Application, Libraries and System (Link-Time)</td>
</tr>
<tr>
<td><strong>Linux Application</strong></td>
<td>Queues, Signals, Pipes, Intra-Process Shared Memory, Shared System Memory</td>
<td>Semaphores, Mutexes</td>
<td>Process and Threads APIs</td>
<td>Local Process, Static and Shared Libraries</td>
</tr>
<tr>
<td><strong>Linux Driver</strong></td>
<td>Shared System Memory, Read/Write Process Memory</td>
<td>Kernel Semaphores, Spinlocks</td>
<td>Kernel Threads, Tasklets</td>
<td>Full Kernel</td>
</tr>
<tr>
<td><strong>Linux Module</strong></td>
<td>Shared System Memory, Read/Write Process Memory</td>
<td>Kernel Semaphores, Spinlocks</td>
<td>Kernel Threads, Tasklets</td>
<td>Module-Local and Exported Kernel Symbols</td>
</tr>
</tbody>
</table>

Two important distinctions are called out in Table 1:
- RTOSes are egalitarian, letting application and I/O code touch any address and perform almost any activity, whereas Linux is much more hierarchical and restrictive.

- Legacy RTOS code can see every symbol or entry point in the system, at least at link time, whereas Linux user code is isolated from and built separately from kernel code and its accompanying namespace.

The consequences of the Linux hierarchy of privileged access is normally only kernel code (drivers) actually accesses physical memory. User code that also does so must run as root.

In general, user-space code is isolated from the Linux kernel and can see only explicitly exported symbols as they appear in /proc/ksyms. Moreover, visible system calls to the kernel are not invoked directly but by calls to user library code. This segregation is intentional, enhancing stability and security in Linux.

When you write a driver, the opposite is true. Statically linked drivers are privy to the entire kernel namespace, not only exports, but have zero visibility into user-space process-based symbols and entry points. And, when you encapsulate driver code in run-time loadable modules, your program can leverage only those interfaces explicitly exported in the kernel by the *EXPORT_SYMBOL* macro.

**Migrating Network Drivers**

As indicated above, porting character and block device drivers to Linux is a straightforward if time-consuming activity. Porting network drivers, though, can seem much more daunting.

Remember that while Linux grew up with TCP/IP, most RTOSes had networking grafted onto them in the late 1990s. As such, legacy networking often only presents bare-bones capabilities, such as being able to handle only a single session or instance on a single port or to support only a physical interface on a single network medium. In some cases, networking architecture was generalized after the fact, as with Wind River VxWorks MUX code to allow for multiple interfaces and types of physical connection.

The bad news is that you likely have to rewrite most or all of your existing network interfaces. The good news is that re-partitioning for Linux is not hard and you have dozens of open-source network device driver examples to choose from.

Your porting task is to populate the areas at the bottom of Figure 4 with suitable packet formatting and interface code.
Writing network drivers is not for beginners. Because, however, many RTOS network drivers actually were derived from existing GPL Linux interfaces, you might find the process facilitated by the code itself. Moreover, there is a large and still-growing community of integrators and consultants focused on making a business of helping embedded developers move their applications to Linux, for reasonable fees.

**Conclusion**

The goal of this article has been to give embedded developers some insight into both the challenges they will face and benefits they will realize from moving their entire software stack from a legacy RTOS to Linux. The span of 2,800 words or so is too brief to delve into many of the details of driver porting (driver APIs for bus interfaces, address translation and so on), but the wealth of existing open-source GPL driver code serves as both documentation and a template for your migration efforts. The guidelines presented here should help your team scope the effort involved in a port of RTOS to Linux and provide heuristics for re-partitioning code for the best native fit to embedded Linux.

As Director of Strategic Marketing and Technology Evangelist when he wrote this article, Bill focused his 17+ years of industry experience on advancing MontaVista and embedded Linux in today's dynamic pervasive computing marketplace. His background includes extensive embedded and real-time experience with expertise in OS, tools, software licensing and manufacturing.