



WHITE PAPER

Moving from a Proprietary RTOS
To Embedded Linux[®]

Prepared by

Bill Weinberg, MontaVista Software, Inc.

Table of Contents

Introduction	4
The Porting Process	4
Architectures for Porting	5
APIs – Applications Programming Interfaces.....	10
IPCs and Synchronization.....	13
API and IPC Accommodation Strategies	16
Other Issues.....	17
Case Studies.....	17
Conclusion	18

Abstract

Whether you are planning a move to embedded Linux in the near future or are just considering the level of investment to convert existing applications to run on embedded Linux, this white paper will help you understand the transition process, to assess the challenges and risks involved, and appreciate the benefits realized from such a move. Specifically, this white paper addresses the porting process, API and IPC conversion, and reliability issues.

Moving from a Proprietary RTOS to Embedded Linux

Bill Weinberg

Introduction

Embedded Linux is rapidly encroaching upon the application spaces once considered the exclusive domain of embedded kernels like Wind River's VxWorks, pSOS, and in-house platforms. Aggressive estimates show embedded Linux and open source garnering up to half of all new designs in the next 24 months¹ while even conservative studies show Linux taking the lion's share (47%) of embedded designs by 2005².

So, whether you are planning a move to embedded Linux in the near future or are just considering the necessary level of investment to convert your existing application to run on embedded Linux, this paper will help you understand the transition process, assess the challenges and risks involved, and appreciate the benefits realized from such a move.

The Porting Process

The process for porting from a proprietary RTOS to embedded Linux is really no different from that needed to move any application across host platforms, although the dependencies are somewhat more involved. Let's start out with a discussion of the basic steps required and subsequently address key dependencies like application architecture, APIs, and IPCs.

Basic steps

Most developers using off-the-shelf RTOS development kits will have a mix of vendor-supplied scripts and makefiles for building and configuring system components and user-developed methods for compiling and linking application code with the kernel and run-time libraries. We will focus on the latter since embedded Linux will take over for the legacy RTOS.

The worst-case port will involve an exhaustive audit of application use of all vendor-supplied APIs, call parameters and global data structures as specified in header files and implemented in libraries. Such an audit will reveal several classes of mapping and equivalence among the calls in the RTOS and those available under Linux:

- Transparent mapping
- Near-transparent mapping
- Easy recode / emulation
- Heavy rewrite required

We will address these cases in more detail below.

¹ LinuxDevices Demographic Survey, December 2000

² Venture Development Corporation: 2000 Embedded Software Strategic Market Intelligence Program

The ideal port would involve applications leveraging only easily mappable calls and so would entail only the substitution or aliasing of key header files and replacement of companion libraries as specified in make and build scripts. Departures from this ideal (a.k.a., *reality*) may result in the need to re-architect and recode.

As it turns out, MontaVista Software (and others) have built RTOS porting and emulation kits that ease the transition by implementing key API translation and emulation functions for the most-used interfaces in kernels including Wind River Systems' VxWorks and pSOS. By leveraging these kits, the porting process is greatly accelerated and streamlined, such that the resulting port takes the form of

1. Set up a Linux-based³ cross development environment including cross development tools (e.g., MontaVista Linux Professional Edition) and RTOS kits (if any)
2. Copy RTOS application source tree to development environment
3. Modify build scripts to link emulation kit libraries (e.g., `kernelLib.o` `taskLib.o`) or use makefiles supplied with the kits
4. Modify/alias pathnames and/or modify source files to reference substitute header files (original RTOS header files can introduce conflicts with native Linux headers)
5. Add `#includes` for new Linux header files either embedded in emulation headers or to your application sources themselves (usually `stdio.h`, `stdlib.h`, `string.h`, `unistd.h`, and `errno.h`)
6. Attempt to make/build and examine results
7. FIRST resolve symbolic issues for implemented APIs (e.g., simple naming and type-safe linkage issues)
8. Address unimplemented APIs and data structures (see below)
9. Repeat steps 5. -7. as needed (a.k.a. "whack-a-mole")
10. Tune performance, as needed
11. Selectively recode and re-architect to leverage native Linux constructs

Let's now examine the challenges you are likely to face at each step of process and in general as you move off your RTOS and onto embedded Linux.

Architectures for Porting

The basic architecture of an RTOS-based application has changed little in the last 20 years, despite huge advances in microprocessors and other aspects of hardware design. RTOS applications are structured as a set of tasks (C functions, typically), statically linked to run-time libraries (including the RTOS kernel itself). These tasks reside and execute in a single physical address space (in RAM or sometimes in ROM) that they share with each other and with global application data, system data, application and kernel stacks, memory-mapped I/O ports, and the RTOS kernel itself.

Classic RTOS Model – Maximum Exposure

This time-worn and familiar architecture, while simple, is highly exposed to corruption: runaway tasks can write over application code and data, accidentally write into peripheral device registers,

³ The process is possible under Windows or another host, but using Linux for cross development greatly eases key jobs like building target embedded Linux file systems and device nodes

and can corrupt kernel data structures and overwrite the kernel code. Tightly packed task stacks can easily underflow and overwrite one another, or charge downward through memory to corrupt the top of the heap or other data or code laid out nearby.

Granularity of Failure in Space and Time with an RTOS

At a higher level, this informally organized and highly exposed architecture presents two key challenges to code quality: scope of the failure itself, and 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 that it failed at all. Thus, even when a failure is detected and recovery attempted, the granularity of failure ends up being the entire system: monitor code cannot usually safely restart tasks 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.

Most often when a program goes awry, it does so “silently”: an errant task can corrupt data and code anywhere in the 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 at a later date – seconds, hours, or months later. When aberrant symptoms do appear, it will be extremely difficult to associate unexpected program behavior, whether subtle or “crash-and-burn”, with the original cause.

The Linux Programming Model

Linux, as UNIX-compatible operating system, presents a much more robust application and system programming model to the programmer. Applications execute in their own protected address spaces, for the most part invisible to one another, and are prevented from overwriting their own code through the use of hardware-based memory management units (MMUs) present on most modern 32 and 64 bit processors.

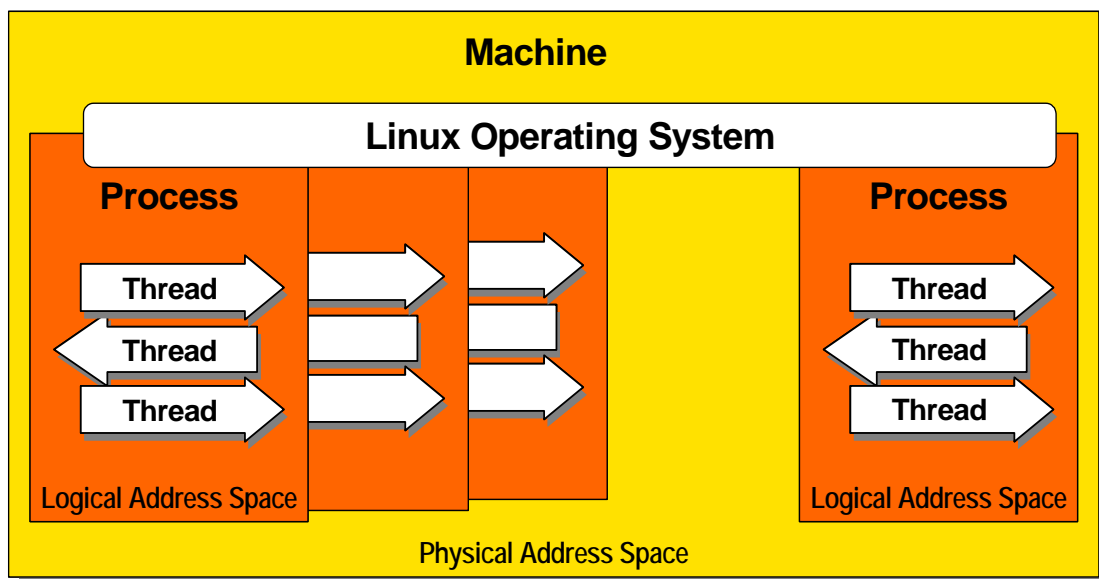


Figure 1. — Multiple Processes and Threads Running Under Linux

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.

Granularity of Failure and Rapid Recovery with Embedded Linux

Because each application process is self-contained and “sealed off”, most failures are thereby limited in scope to an individual process, which receives the “segmentation violation” signal (SIGSEGV). These errors include

- Attempts to write to read-only segments (application and kernel code)
- Accesses to uninitialized or misprogrammed pointers
- Stack underruns

By default, the programs receiving a SIGSEGV terminate, but user handling and remedies are possible with the appropriate signal handler.

When a process does fail, its resources (RAM, open files, sockets, IPCs, etc.) are completely recovered by the operating system, staunching memory leaks and permitting the clean restart of the failed process without need to reboot the system. Moreover, the “silent” corruption that can occur in an RTOS surfaces *immediately* with embedded Linux, with failed processes optionally leaving behind a core file readable by standard debuggers to find the source of corrupting operations.

The First Order Port

The above architecture descriptions readily suggest a very 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, i.e., entire system memory complements, map into Linux virtual address spaces – a multi-board or multiple processor architecture (like a VME rack) migrates into a multi-*process* Linux application⁴, as in Figure 2. below.

⁴ Such migrations usually entail a CPU upgrade as well.

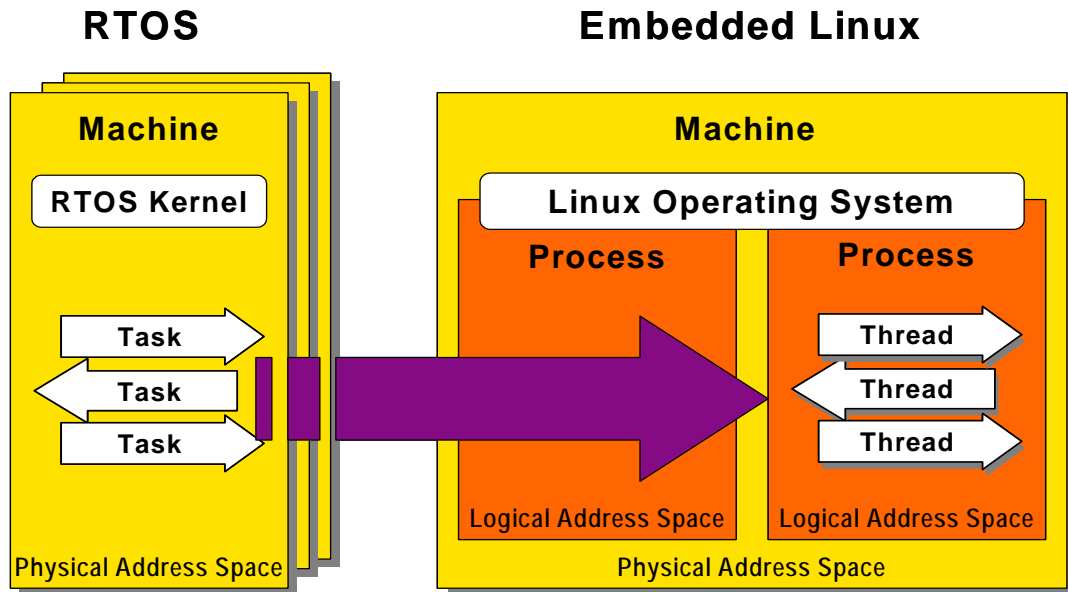


Figure 2. – Migrating RTOS Tasks in a Physical Address to Linux Multi-Threaded Processes

Architectural Considerations – Process & Thread Creation Overhead

Whether you use RTOS emulation kits for Wind River VxWorks and pSOS, or perform the port unaided, you will ultimately have to make decisions regarding whether to implement RTOS tasks as processes or as threads. While at its heart, the Linux kernel treats both processes and threads as co-equal for scheduling purposes, there are different APIs for creating and managing each type of entity, and performance and resource costs (and benefits) associated with each.

In general, processes are “heavier” than threads because they carry more context with them. A Linux thread context (like an RTOS task) consists primarily of a subset of CPU registers, a stack, a current program counter (PC), and some entries in the kernel’s data structures (TCBs, in an RTOS). A process adds a complete virtual address space to this definition. 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 weighty process context comes at two key junctures: process creation time and inter-process context switch time.

Process vs. Thread Creation

RTOS code strives for lightweight execution whenever possible. As such, while many RTOSes 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 process creation is not intentionally a more cumbersome operation – Linux processes are heavier because they offer greater benefits of protection and reliability. The kernel mechanism for creating processes is the `fork()`⁵ system call.

⁵`fork()`, `vfork()`, etc.

Forking New Processes

Unlike task and thread creation, which essentially identify existing program functions as new schedulable entities (as in VxWorks task creation), `fork()` causes the currently executing to split into two amoeba-like identical copies, a parent and a child. The parent and child initially only differ in the PID (Process ID), so the first thing programs do after a `fork` is to ponder, existentially, who am I? This deliberation is accomplished most often with a `switch` statement in C. The return value of `fork()` for the parent will be the child's PID, whereas the child will see the return as 0. Thus, the parent can “watch over” the child and each “knows” its identity.

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

pid_t new_PID;

new_PID = fork();

switch (new_PID) {

    case 0 :
        /* child code runs here */
        printf("I am the child -- my PID is ??\n");
        break;

    case -1 :
        /* oops - something went wrong */
        exit( errno );
        break;

    default :
        /* parent code runs here */
        printf("I am the parent -- my child's PID is %d\n", new_PID);
        break;
} /* switch */
```

Figure 3. – Listing of unified parent / child code with `fork()` call

Forking involves several steps (simplified):

- Create new virtual address space
- Map TEXT pages (no copying – image is shared)
- Copy DATA pages (actually occurs per page, on first write)
- Create copies of all current file descriptors
- Create scheduler entry (with `clone()`)
- Assign new PID
- Schedule child process
- Run parent/child code or call `exec()`⁶ to replace image

⁶ Actually, the `exec()` family – `execv()`, `execle()`, etc.

The last step, optionally calling `exec()`, offers the option of encapsulating child code and functionality into the same program as the parent or of “starting over” and loading in a new binary from a file system path.

Thread Creation

Thread creation with the `clone()` system call or the `pthread_create()` API is altogether a simpler affair, since all threads within a process share the same address space, file descriptors, etc.

- Layout new stack in current user process space
- Create scheduler entry
- Assign new ID
- Schedule new thread or wait per semantics of pthreads interface

Context Switch Implications

Switching among threads and processes involves different amounts of effort and context saving. The fastest context switch is of course among threads running in a single process-based virtual address space. Switches between processes involve TLB spills, reloading of page translation table entries, and potential saves/restores of additional context like FPU, MMX, AltiVec and ARM co-processor registers⁷.

Design Criteria – Processes or Threads

While a first order port will typically map RTOS tasks onto Linux threads, subsequent modifications will require decisions on the part of the developer. Following are some heuristics for making this decision:

- In general, create processes during initialization and threads on the fly
- Use processes for greater reliability and where health monitoring and failure detection (via SIGCHLD) is a concern
- Use processes to encapsulate third-party code; if it blows up, it can do much less harm and can always be restarted
- No universal benchmarks are available to compare process and thread creation costs. Calls to *fork* can run into tens or hundreds of milliseconds; *cloning* is much more sprightly and executes in tens of microseconds.
- Use of `exec()` carries the heaviest cost, since it accesses file systems to load an executable image and must create a new virtual address space

APIs – Applications Programming Interfaces

While the benefits of moving to a Linux process-based programming model are enticing, you still have to address the particulars of moving your application’s use of RTOS programming interfaces over to the repertoire offered by Linux. The good news is that Linux features perhaps the richest array of APIs of any embedded operating system; the bad news is that your code may exploit RTOS calls and features that do not readily translate into the Linux model.

⁷ Moreover, some kernel implementations treat even intra-process thread context switch like inter-process switches, spilling the TLBs, with comparable overhead.

Task Management	Queues / Events	Semaphores	Partitions	Timers
t_create()	q_broadcast()	sm_create()	pt_create()	tm_wkafter()
t_delete()	q_create()	sm_delete()	pt_delete()	
t_getreg()	q_delete()	sm_ident()	pt_getbuf()	
t_ident()	q_ident()	sm_p()	pt_ident()	
t_mode()	q_receive()	sm_v()	pt_retbuf()	
	q_send()			
t_resume()	q_urgent()			
t_setpri()	q_vcreate()			
t_setreg()	q_vdelete()			
t_start()	q_vident()			
t_suspend()	q_vreceive()			
	q_vsend()			
	q_vurgent()			
	q_vbroadcast()			
	ev_receive()			
	ev_send()			

Figure 4 –Wind River pSOS APIs Emulated for MontaVista Linux

Vendors used to characterize their kernels in terms of the number and type of systems calls, or direct OS interfaces, usually implemented by traps or exceptions (or by subroutine calls in some RTOSes, like VxWorks). In reality, application code accesses these system calls via libraries that either act as “wrappers” for the system calls or may implement the entire functions in library code without ever calling the kernel. Examples of the first case are task creation and scheduling calls, like VxWorks `taskInit()`, while examples of the second include library-based threading schemes on older UNIX systems or so-called “green threads” in Java.

Task Management		Message Queues	Semaphores	Watchdogs
taskSpawn()	taskSafe()	msgQCreate()	semGive()	wdCancel()
taskInit()	taskUnsafe()	msgQDelete()	semTake()	wdCreate()
taskActivate()	taskDelay()	msgQSend()	semFlush()	wdDelete()
taskDelete()	taskName()	msgQReceive()	semDelete()	wdStart()
taskDeleteForce()	taskNameToId()	msgQNumMsgs()	semBCreate()	
taskSuspend()	taskIdVerify()		semCCreate()	
taskResume()	taskIdSelf()		semMCreate()	
taskRestart()	taskIdDefault()		semMGiveForce()	
taskPrioritySet()	taskIsReady()			
taskPriorityGet()	taskIsSuspended()			
taskLock()	taskTcb()			
taskUnlock()	taskIdListGet()			

Figure 5. – Wind River VxWorks APIs Emulated for MontaVista Linux

In actual practice, your application probably makes no distinction between system calls and library functions and may leverage dozens or even hundreds of available APIs under an RTOS or Linux. Kernels like VxWorks, pSOS, VRTX, Nucleus, and other RTOS products have accrued over one thousand APIs (cf. the Wind River “big blue books”) in their decades of commercial existence and it is not practical to address the mass of those APIs herein nor in the emulation/porting kits mentioned above. A more pragmatic approach is to translate and emulate a

clean core set of the four or five dozen most common calls, and to leave the rest for *ad hoc* translation and implementation (see Figures 4. and 5.).

Indeed this is the approach followed by MontaVista in its VxWorks and pSOS kits.

Key Standard APIs and Libraries

Building applications that leverage standard, common APIs achieves two complementary purposes: it allows code to be ported *to* standards-based operating systems (like Linux), and it allows that same code later to be ported *from* such an environment, more easily than with proprietary APIs. Indeed, while some commercial RTOSes include standard call sets from POSIX or BSD, they exist as “window dressing” and are seldom employed. Rather, the proprietary, closed APIs particular to a given kernel are the most used, and it is these that have traditionally locked projects into long-term commitments to a particular platform or solution.

If you are porting standards-based code, whether implemented over an RTOS (like the VxWorks POSIX.1b subset or the VRTX POSIX subset APIs), or written for a different flavor of UNIX, or if you are considering which of several API options to choose for new code, it is important to understand a few of the most common standards in use under Linux and other open systems.

POSIX

The Portable Operating System Interface, POSIX (pronounced like *positive*, not like *posey*), prevalent in the UNIX-based open systems community and in government and military arenas, has had limited impact on the traditionally closed and proprietary world of real-time embedded operating systems. The POSIX family of standards was originated by NIST (the US-based National Institute of Standards and Technology⁸), but now falls under the auspices of the IEEE⁹ as IEEE1003 and other standards. POSIX standards of interest include:

POSIX.1

This standard defines the core of UNIX-type operating systems, including key basic concepts such as process definition, file system, basic I/O, and core APIs. Two important notions exist with regard to POSIX.1: compliance and conformance. Compliance implies that a given OS platform implements some sub-set of the standard and that the implementation is documented. Even platforms implementing a trivial subset can be termed *POSIX compliant*. *Conformance*, conversely, is a much stricter criterion, meaning that an OS has been subject to a certification test and passed within allowable parameters.

Embedded Linux, as an instance of UNIX (a UNIX *work-alike*), is POSIX-compliant operating system. Although conformance is technically an all-or-nothing benchmark, versions of embedded Linux anecdotally appear to be between 25% and 50% conformant against NIST test suites¹⁰, depending upon architecture¹¹. Few if any RTOSes approach even 10% of POSIX.1, but do implement subsets of POSIX.1b and .1c.

⁸ See <http://www.nist.gov> for additional information

⁹ Institute of Electrical and Electronics Engineers – see <http://www.ieee.org/> for general information and http://standards.ieee.org/reading/ieee/std_public/description/posix/ for POSIX-related web pages

¹⁰ Test suites available from http://www.itl.nist.gov/div897/ctg/posix_form.htm

¹¹ Many POSIX.1 tests involve the particulars of terminal interfaces and file system constructs not present on deeply embedded architectures.

POSIX.1b

Originally part of POSIX.4, the “real-time extensions”, POSIX.1b focuses on inter-process communications and synchronization, and deals with entities like queues, semaphores, and prioritized signals. It also includes, interestingly, the definition for the `mmap()` call.

Many RTOS platforms, like VxWorks, include POSIX.1b subset interfaces as alternatives to their native IPC/synchronization APIs. Linux in general does not track POSIX.1b (except for `mmap()`), implementing instead the more common SVR4 versions of the same mechanisms.

POSIX.1c

Also originally part of POSIX.4, POSIX.1c details the APIs and semantics for POSIX threads, or pthreads, whose definition is somewhat intertwined with issues from POSIX.1b. Linux implements pthreads as a standard user level library that maps onto the native Linux kernel threading interface (e.g., `pthread_create()` is a wrapper for `clone()`). The pthreads interface is the best analogue for RTOS tasks when porting code to Linux.

POSIX.13

POSIX.13 is a proposed derivative standard for sub-setting other POSIX standards. It is only germane in that the Embedded Linux Consortium has considered it for inclusion in its own standardization efforts. It primarily exists to allow non-UNIX/Linux platforms to masquerade as POSIX compliant.

SVR4, BSD, and Other UNIX APIs

SVR4 (UNIX System V, Revision 4) and versions of the Berkeley Software Design-derived BSD UNIX¹² are prevalent de facto system standards that greatly influence Linux. That is, Linux implements large subsets of those UNIX APIs (e.g., the Linux `ipc()` system call for SVR4 shared memory, queues, and semaphores and the BSD sockets calls and TCP/IP stack).

If you are familiar with SVR4, BSD, or other common UNIX implementations (like AIX, HP-UX, etc.), you'll feel right at home on Linux.

C Language Libraries

Many of the APIs in embedded designs, RTOS-based or otherwise, are simply standard C language libraries that either directly implement functionality or act as wrappers for system calls. On Linux, you'll find `libc/glibc` completely familiar, although larger in scope and more comprehensive than RTOS implementations (like `pREPC` for `pSOS`).

IPCs and Synchronization

Every operating system, whether general-purpose or embedded, addresses the challenge of supporting inter-task communication and synchronization in a slightly different way. The good news is that the most common set of IPC (inter-process communication) mechanisms found in RTOS repertoires have ready analogues in embedded Linux (indeed, Linux is extremely rich in this area). The bad news is that the mapping is seldom completely one-to-one and that even when apparent IPC equivalents exist, their scope may be focused on communications among

¹² See <http://www.bsd.org>

processes rather than among lighter-weight threads most analogous to RTOS tasks, with subtly differing semantics.

The following sections survey the most common RTOS IPCs and how they map onto Linux analogues, as summarized in Figure 6.

RTOS IPCs	Linux IPCs
Semaphores (Counting and Binary)	SVR4 Semaphores
Mutexes	POSIX.1c Mutexes, Condition Variables
Message Queues and Mailboxes	Pipes/FIFOs, SVR4 queues
Shared Memory	Shared Memory
Events and RTOS Signals	Signals, RT Signals
Timers, Task Delay	POSIX timers/alarms <code>sleep()</code> and <code>nanosleep()</code>
Watchdogs, Task Regs, Partitions/Buffers	Emulated by tool kits

Figure 6. – RTOS and Linux IPC and Synchronization Mechanisms

Mechanisms

Following is a brief discussion of the mapping of RTOS IPCs and synchronization mechanisms onto their Linux equivalents as implemented for C language programming. The focus herein is on mechanisms supplied through core Linux system calls and common libraries. The world of Linux and open source is vast, however, and many additional options exist as patches and add-on libraries from other sources.

Semaphores and Mutexes

Linux offers two levels of compatibility with RTOS synchronization and mutual exclusion mechanisms. In general the SVR4 semaphores are the most robust and most used, especially for synchronization among Linux processes. In contrast, the POSIX.1c interface offers mutex/condition variable-based synchronization and mutual exclusion. It is important to note that while most RTOSes make a point of lightweight implementation of semaphores (and even benchmark semaphore creation), Linux use of SVR4 semaphores is somewhat “heavier”, in that the interface performs additional parameter checking.

Queues and Mailboxes

Most RTOSes offer lightweight queuing constructs or mailboxes for passing discrete messages and sometimes light payloads among tasks. Linux has a rich repertoire of message-passing capabilities, starting with named pipes and FIFOs, and including SVR4 queues. POSIX.1b *mqueues* are not implemented in favor SVR4 queues (from which POSIX.1b queues are derived, anyway).

Shared Memory

Most RTOS-based programs make use of informally shared data structures. Some RTOSes offer a thin formalism for sharing blocks of memory, but without MMU-based memory protection, such schemes are artificial at best and unneeded overhead at worst.

Linux, with its POSIX process model and strictly enforced virtual addressing, offers applications a robust shared memory capability. Processes can share memory with each other and with drivers via calls like the POSIX.1b `mmap()` and the SVR4 `shmget()` interfaces, and govern the usage with the synchronization and mutual exclusion mechanisms described above.

Shared memory among boards in a multiprocessor system, like a VME cage, is one case where many RTOSes did and do add value with shared memory mechanisms (like VxWorks VxMP). Again, the Linux based mechanisms leverage the `mmap()` API, allowing for shared memory across VME, CompactPCI or other interconnects to be mapped into Linux virtual address spaces.

Events and RTOS Signals

While some RTOSes implement events as message-bearing queues, in general events are asynchronous mechanisms that carry no payload. As such, RTOS events translate well to Linux signals, either the standard non-queuing variety (signals 0-31) or the “real-time” asynchronous, prioritizable signals added by POSIX.1b.

Linux makes extensive use of signals for notification. When porting RTOS code to make use of signals, it is absolutely essential to understand the sometimes-complex semantics of Linux signals and their impact on other primitives, like *threads*.

Timers and Task Delays

RTOS-based code makes extensive use of both software and hardware timer implementations, as well as task delay APIs (e.g., VxWorks `taskDelay()`). Linux offers support for large complements of timers and alarms, as well as a variety of delay options.

Timers

RTOS timers translate well into Linux interval timers and alarms (`setitimer()` and `alarm()`) that generate signals on timeout. The key difference between Linux timers and most RTOS timer implementations is that RTOS timer APIs tend to quantify time in terms of RTOS system clock ticks while Linux attempts to use “real” time (seconds, microseconds, or nanoseconds).

The Linux 2.4 kernel is especially adept at managing large numbers of software timers (even thousands of them) with low overhead.

Delays

RTOS task delay calls translate into a family of *sleep* APIs: `sleep()` and `nanosleep()` are program calls that wait an appropriate number of seconds or nanoseconds, and `usleep` is a shell utility that pauses for a specified number of microseconds.

While Linux is quite adept at time management in general, *sleep* calls (and timers too) depend on the resolution of the system clock and so seldom implement down to the putative resolution of their parameters (e.g., nanoseconds).

Periodic Task Execution

While most RTOSes use timers to implement long-term periodic task execution (once per hour, once per day tasks), Linux offers cron for spawning such activities at the process level.

Clock Resolution

Embedded developers will at first glance be alarmed that while Linux offers the above timer and delay interfaces, the clock resolution of those mechanisms is extremely coarse. Since Linux uses its basic preemption clock as a time base for all such calls, the available resolution is effectively 10ms (with longer worst cases). Thus, even carefully specified nanosecond delay counts will be resolved to this coarser granularity.

Several solutions exist for this deficiency. One is to use hardware-based timers and to implement your own interrupt-based delay mechanism (a common technique for timer-impaired RTOSes as well), but one that unfortunately sidesteps the benefits of Linux. Another is to use a sub-kernel (like RTLinux) that may offer finer time bases.

The more interesting option is one that MontaVista Software is pursuing – to offer higher resolution timers in standard Linux. Timer resolution in this scheme will scale with CPU clock speeds, so as not to induce undue overhead (and thrashing) from incessant timer interrupt service. Visit the open source project at <http://sourceforge.net/projects/high-res-timers> for additional information

Watchdog Timers

RTOSes use watchdogs most commonly to enhance system reliability: programmers pepper their code with watchdog timer resets, such that should a watchdog ever expire (time out), it is indicative of a critical fault necessitating a reboot.

While Linux offers other means to increase system robustness, applications may still need watchdog-like functionality on a (ported) per process basis. Timer signal handlers can easily emulate such terminator behavior, and indeed MontaVista supplies this functionality in its RTOS emulation kits.

RTOS-specific IPCs

In addition to watchdogs, MontaVista provides support for RTOS constructs like Task Registers, Partitions, Buffers, etc.(see Wind River VxWorks and pSOS APIs in Figures 4. and 5.).

API and IPC Accommodation Strategies

In this article we have looked at common calls and IPCs for VxWorks and pSOS. Your commercial or in-house RTOS is likely to implement many comparable calls, but is just as likely to feature its own unique APIs and IPCs.

Whatever the platform in question may be, accommodation of its particulars falls into three categories:

Equivalence

Many RTOSes offer calls completely or nearly identical to Linux APIs. Because many RTOSes were written by UNIX programmers, they are likely to feature entry points like *open*, *write*, etc. Such calls will either map 1:1 completely unchanged, will be hidden by compiler library wrappers, or may require some minimal tweaking with *#defines* in header files.

Emulation of APIs

Some RTOS APIs, while not differing greatly, will require “massaging” with the insertion of library code to emulate additional or different functionality. An example is pSOS+ APIs, which carry

notoriously long and obscure parameter lists. Since most programmers only use the first few parameters anyway, you can either nail them down as constants in your emulated code, or encapsulate them in polymorphic C++ class methods¹³.

Emulation can carry performance costs, and developers always assume that emulated code runs less efficiently than the original native construct. Anecdotally, my company's user base (see Case Studies below) have experienced only performance increases, both in general performance and in the area of networking. Your mileage may vary!

Recoding

When RTOS constructs simply don't exist for Linux, neither natively nor via emulation libraries, you will have to recode and re-implement. While recoding is usually the minority case, it is the least pleasant!

Other Issues

This article has focused primarily on APIs and IPCs, and porting architectures. Other interesting RTOS migration issues not targeted herein (but addressed by MontaVista Linux) include

- Linux real-time performance (including native real-time vs. kernel substitution, and preemption)
- Scheduling priority and policies
- Scaling Linux for embedded resource footprints
- Booting embedded Linux
- Embedded Linux spindle-less file systems (RAM, CramFS, JFFS, etc.)

Case Studies

In its two-year history, MontaVista Software has engaged with over 250 customers on almost 300 designs. Among these designs have been several dozen that migrated specifically from legacy RTOS code, and among those many have benefited from the existence of the RTOS emulation kits, while others have migrated from other embedded platforms.

Cyclades

Cyclades manufactures networking products for the enterprise Linux marketplace, specifically terminal servers, multi-channel serial interfaces and routers. They had been maintaining a proprietary RTOS and TCP/IP stack for their MPC860T design for as long as ten years, at great expense. Their conversion to embedded Linux required approximately five months, to go from initial effort to shipping product.

For more information¹⁴, see the customer profile at <http://www.mvista.com/dswp/Cyc22.pdf>

¹³ See Weinberg and Letheby [1992].

¹⁴ See also Saito [2001]

Get Engineering

Get Engineering is a products and consulting firm that provides military networking equipment to the U.S. Navy via prime contractors. Their team of three hardware engineers and two software engineers moved approximately 15,000 lines of C code for VME-based PowerPC processors from Wind River VxWorks to MontaVista Linux in eight weeks time (they had allocated up to nine months for the project). The application, which exhibited a variety of real-time requirements, was able to leverage the VxWorks to pthreads conversion kit for the vast majority of the code employed – only one or two APIs were missing out of the dozens used in their application, and no recoding was needed for their port.

For additional information, see article at <http://www.mvista.com/news/2001/get.html>

Interface Concept

Interface Concept builds high performance networking equipment for VME, PCI and CompactPCI form factors, based on Motorola MPC860 and MPC8260 processors. They reported being “delighted by the performance improvement that MontaVista Linux delivered in benchmark tests, compared with proprietary VRTX and pSOS real time operating systems that ran on previous generation Ethernet switches,” with “between a five-fold and ten-fold improvement in TCP/IP throughput”.

For additional information, see profile at <http://www.mvista.com/dswp/InterConcept.pdf>

Conclusion

The move is on – developers are leaving behind first generation RTOSes in search of more reliable and open embedded platforms like Linux. While the migration from these traditional systems does present a variety of challenges, the benefits far outweigh the investment needed to move to embedded Linux. The risk doesn't arise from leaving behind your familiar environment, tools, and APIs – the real risk lies in standing still while the embedded and pervasive systems development communities move forward, at Internet speed.

By following the steps outlined above, and by leveraging tools like the MontaVista RTOS migration kits, you can successfully migrate your existing legacy RTOS code to a modern embedded Linux platform.

References

- Bovet, Daniel, and Cesati, Marco [2001]. *Understanding the Linux Kernel*. O'Reilly & Associates; ISBN 0-596-00002-2.
- Gallmeister, Bill. O. [1995]. *POSIX.4 : Programming for the Real World*. O'Reilly & Associates; ISBN: 1565920740.
- Robertson, Gary (for MontaVista Software). [2000]. *Porting pSOS+ Applications to Linux: A Brief Tutorial on RTOS Legacy Code and Embedded Linux*. MontaVista Software, Inc.
- Robertson, Gary (for MontaVista Software). [2000]. *Porting VxWorks Applications to pthreads: A Brief Tutorial on RTOS Legacy Code and Embedded Linux*. MontaVista Software, Inc.
- Saito, Marcio. [2001]. "Embedded Linux at Cyclades Corporation: Hesitating in switching from your proprietary RTOS to Linux? Don't". In *Embedded Linux Journal*, July (Issue #4).
- Weinberg, William, and Letheby, Nick. [1992] "A C++ Interface for Real-time Multi-tasking Operating Systems". In *Proceedings of the Embedded Systems Conference*.
- Wind River Systems. [1995]. *VxWorks Programmers Guide*.

Corporate Headquarters

United States

MontaVista Software, Inc.
1237 East Arques Ave.
Sunnyvale, CA 94085
Tel : (408) 328-9200
Fax : (408) 328-9204
email: sales@mvista.com
<http://www.mvista.com>

European Headquarters

The Netherlands

MontaVista Software BV Maarssen
Tel: +31 (0) 346581090
email: info-be@mvista.com

Belgium

MontaVista Software Hove
Tel: +32 (0)474 53 09 05
email: info-be@mvista.com

France

MontaVista Software SAS
Tel: +33 (0)1 30 16 28 28
email: info-fr@mvista.com

Germany

MontaVista Software GmbH
Tel: +49-89-375 90
email: info-de@mvista.com

Sweden

MontaVista Software AB
Tel: +46 8 527 570 00
email: info-se@mvista.com

UK

MontaVista Software Limited
Tel: 08709 010870
email: info-uk@mvista.com

Japan Headquarters

Japan

MontaVista Software Japan, Inc.
Tel: +81-3-5469-8840
email: info-jp@mvista.com

Asia Pacific Headquarters

Hong Kong

MontaVista Software, Inc.
Tel: +852 2506 6201
email: info-ap@mvista.com

Singapore

MontaVista Software Singapore Pte. Ltd.
Tel: +65 8385442
email: info-ap@mvista.com

