white paper

Moving Legacy Applications to Linux: RTOS Migration Revisited

Version 2.1– July 29, 2007

By William Weinberg

Table of Contents

Abstract

Whether you are planning a move to embedded Linux or are just considering the investment needed to convert existing applications to run on embedded Linux, this white paper will help you understand the transition process, assess the challenges and risks, and appreciate the benefits realized from such a move.

This white paper addresses how to map legacy architectures onto Linux, options for migrated application execution, API and IPC translation, enhanced reliability realized from migration, the migration process itself, and application-specific migration challenges and solutions.

Introduction

Embedded Linux is rapidly encroaching upon application spaces once considered the exclusive domain of embedded kernels like VxWorks, pSOS, and in-house platforms. Industry analysts show 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 . So, whether you are planning a move to embedded Linux or are just considering the investment needed 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.

Migration Execution Architectures

While Linux increasingly takes the place of traditional RTOSs, executives, and kernels, the architecture of the Linux operating system is very different from legacy OS architectures. Moreover, there exists more than one means to host legacy RTOS-based applications on a POSIX-type OS like Linux. The following section lays out three approaches to migration, from conservative means that preserve legacy attributes and architecture to more extensive revamping of code and application structure.

Emulation, Virtualization, and Native

This section compares and contrasts the three most relevant migration and re-hosting paradigms for legacy software under Linux:

1. RTOS API emulation over Linux
2. Run-time partitioning with virtualization
3. Full native Linux application port

RTOS Emulation over Linux

For legacy applications to execute on Linux, some mechanism must exist to service RTOS system calls and other APIs. Many RTOS entry points and stand-alone compiler library routines have exact analogs in Linux and the glibc run-time library, but not all do. Frequently new code must intervene to emulate missing functionality. And even when analogous APIs do exist, they may present parameters that differ in type and number.

A classic RTOS can implement literally hundreds of system calls and library APIs. For example, VxWorks documentation describes over one thousand unique functions and subroutines. Real-world applications typically use only a few dozen RTOS-unique APIs and call functions from standard C/C++ libraries for the rest of their (inter)operation.

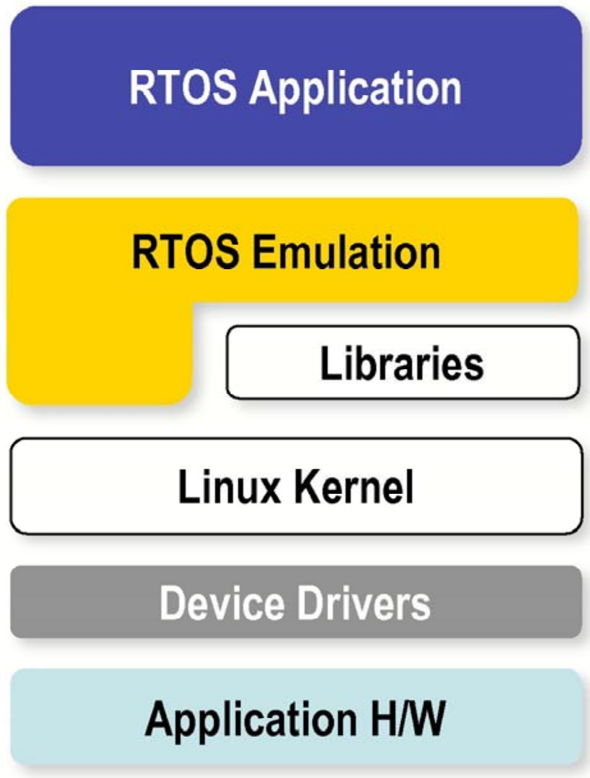

Figure 1. – RTOS emulation over Linux

To emulate these interfaces for purposes of migration, developers only need a core subset of RTOS calls. Many OEMs choose to build and maintain emulation lightweight libraries themselves; others look to more comprehensive commercial offerings from vendors such as MapuSoft. There also exists an open source project called v2lin that emulates several dozen commonly used VxWorks APIs. Learn more at http://sourceforge.net/projects/v2lin/

Partitioned Run-time with Virtualization

Virtualization involves the hosting of one operating system running as an application “over” another virtual platform, where a piece of system software (running on “bare metal”) hosts the execution of one or more “guest” operating systems instances. In enterprise computing, Linux-based virtualization technology is a mainstream feature of the data center, but it also has many applications on the desktop and in embedded systems.

Data center virtualization enables server consolidation, load-balancing, creating secure “sandbox” environments, and legacy code migration. Enterprise-type virtualization projects and products include the Xen Hypervisor, VMware and others. Enterprise virtualization implements execution partitions for each guest OS instance, and the different technologies enhance performance, scalability, manageability and security

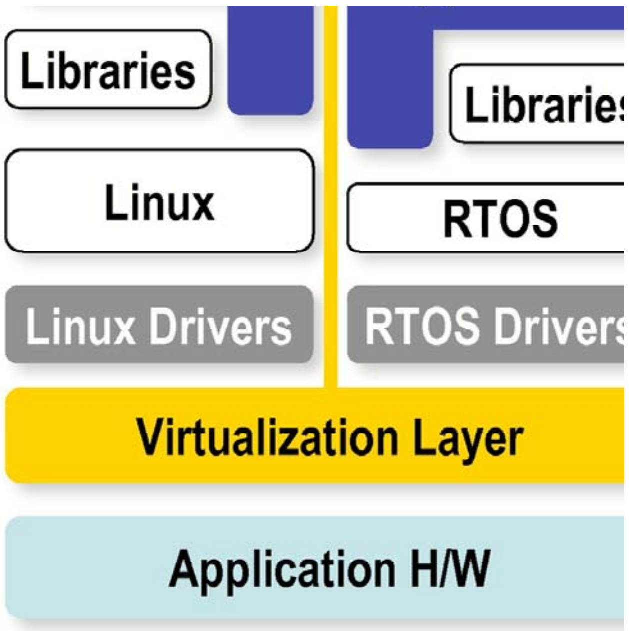

Figure 2. – Partitioned Run-time with Virtualization

Embedded virtualization entails partitioning of CPU, memory and other resources to host an RTOS and one or more guest OSs (usually Linux), to run higher-level application software. Virtualization supports migration by allowing an RTOS-based application and the RTOS itself to run intact in a new design, while Linux executes in its own partition. This arrangement (see Figure 1.) is useful when legacy code not only has dependencies on RTOS APIs but on particular performance characteristics, for example real-time performance or RTOS-specific implementations of protocol stacks. Embedded virtualization as such represents a short and solid bridge from legacy RTOS code to new Linux-based designs, but that bridge exacts a tollOEMs will continue to pay legacy RTOS run-time royalties and will also need to negotiate a commercial license from the virtual machine supplier.

A wide range of options exist for virtualization, including the mainstream KVM (Kernel-based Virtualization Manager) and Xen. Embedded-specific paravirtualization solutions are available from companies like Virtual Logix. (Visit http://www.virtuallogix.com for more information.) Open source options include the L4 partitioned microkernel. (Learn more at http://l4ka.org/)

Native Linux Port of Application

Emulation and virtualization can provide straightforward migration paths for prototyping, development, and even deployment of legacy RTOS applications running on Linux. They have the drawback, however, of including additional code, infrastructure, and licensing costs. Instead, “going native” on Linux reduces complexity, simplifies licensing, and enhances portability and performance.

The choice need not be exclusive. The first time OEMs approach migration they are likely to leverage emulation and virtualization technologies. With greater familiarity with development tools and run-time attributes of Linux, OEMs can re-engineer legacy applications incrementally for native Linux execution. One approach is to choose individual legacy programs for native migration and to host them under Linux in separate processes. This technique works best with software exhibiting minimal or formalized dependencies on other subsystems. Another sensible practice is to implement new functionality only as native code, even if employing emulation or virtualization.

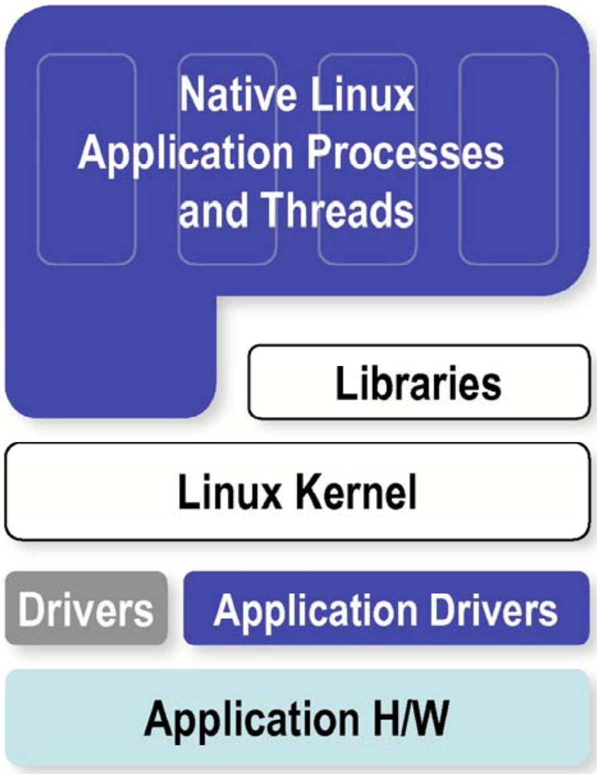

Figure 3. – Native Port of RTOS Application

Mapping Legacy Constructs onto Linux

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 application as in Figure 4. below.

Architectural Considerations: Process and Thread Creation

Whether you use RTOS emulation kits for Wind River VxWorks and pSOS, or perform your 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.

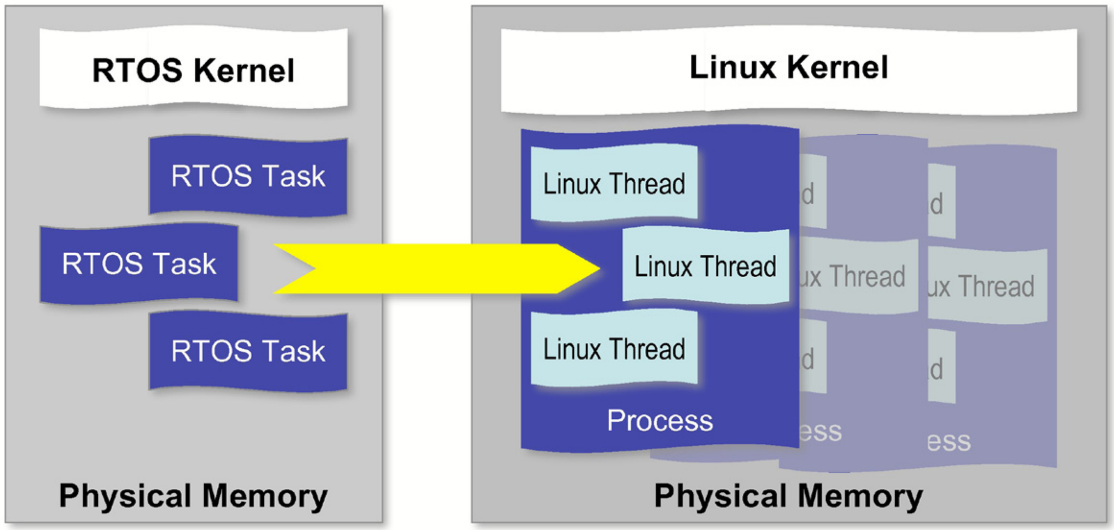

Figure 4. – Mapping RTOS tasks onto Linux threads

In general, processes are “heavier” than threads because they carry more context. 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 weightier process context comes at two junctures: process creation time and inter-process context switch time.

RTOS code strives for lightweight execution whenever possible. As such, many RTOSs offer dynamic task creation APIs, but others feature only static task definition tables, and all RTOS vendors discourage frivolous and frequent task creation to save time and space. The migration process provides a good opportunity to audit task/thread inventory of legacy RTOS applications and to optimize resource usage.

The kernel mechanism for creating processes is the fork() system call. Linux process creation is not intentionally a more cumbersome operation – Linux processes are heavier because they offer greater benefits of protection and reliability.

Forking New Processes

RTOS task and thread creation in both RTOSes and Linux essentially identify existing program functions as new schedulable entities (as in VxWorks task creation). By contrast, the Linux system call/API fork() causes the currently executing file to split, amoeba-like, into identical copies, a parent and a child. The parent and child initially only differ in their 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 5. – Listing of unified parent / child code with fork() call

Forking involves several steps (simplified):

1. Create new virtual address space.
2. Map TEXT pages into new space (no copying – image is shared).
3. Copy DATA pages (actually occurs per page, on first write).
4. Create copies of all current file descriptors.
5. Create scheduler entry (with clone()).
6. Assign new PID.
7. Schedule child process.

The child process can then run “as is” – in the image of the parent, or the child can call execv() to load in a new binary image from a file system path into the child process memory space.

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.

Creating new threads proceeds as follows:

1. Lay out new stack in current user process space.
2. Create scheduler entry.
3. Assign new ID (TID).
4. 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. Switching between threads across process boundaries involves TLB (Translation Look-aside Buffer) spills, reloading of page translation table entries, and potential saves/restores of additional context such as FPU, MMX, Altivec, and ARM co-processor registers.

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.
- Employ processes to encapsulate third-party code; if that code 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.
- Creating entirely new processes / loading new programs (via calls like execv()) carries the heaviest cost, since it accesses file systems to load an executable image and must create a new virtual address space.

Threads Are Not Free

Use of threads in Linux is not “free”. The cost just appears in a different performance budget. Much of the GNU/Linux operating system is architected for multi-processing rather than multi-threading. Commonly used resources like glibc, X11, samba, etc. offer “thread-safe” versions or have been enhanced for thread safety, but at a price.

Additional synchronization and reentrancy needed to support multi-threading and implicitly shared data (but not required by process-based calls) can add overhead and impact performance. For this very reason, samba and other services often appear in multiple versions (multi-process, multi-threaded, and single-threaded).

There is no available metric for comparing the impact of multi-thread threaded versus process-based architectures. You will have to establish the optimal trade off for your particular application.

The Porting Process

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

Considerations

Most developers using off-the-shelf RTOS development kits have a mix of vendor-supplied scripts, IDE configurations, 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. This white paper 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. Most companies maintain documentation describing some portion of their own APIs and API usage. To explore undocumented use, and to audit third-party code, tools like Klocwork K7 may be useful.

A detailed code and API audit will reveal several classes of mapping and equivalence among calls to an RTOS and those available under Linux:

- Transparent mapping: function name, prototype, parameters, and types are identical; semantics may still diverge.
- Near-transparent mapping: prototype mostly the same; API exhibits minor differences in parameters or types.
- Easy recode/emulation: nominally equivalent function/API exists; parameters can be typecast or call directed through a stub or wrapper.
- Heavy rewrite required: no semantic equivalent or one-to-many mapping of functionality.

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.

Basic Steps

Whatever the particulars of your legacy code base, you and your team will likely follow these elementary steps:

1. Set up a Linux-based cross development environment including cross development tools (e.g., MontaVista Linux Professional Edition with DevRocket).
2. Copy RTOS application source tree to development environment.
3. Modify build scripts and IDE configurations to link emulation libraries (if any).
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 Linux header files to your application sources themselves (usually stdio.

h, stdlib.h, string.h, unistd.h, and errno.h) or via emulation headers (if any).

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 5-8 as needed (a.k.a. "whack-a-mole").
10. Tune performance, as needed, using tools and capabilities found in MontaVista DevRocket.
11. Selectively recode and re-architect to leverage native Linux constructs.

Rearchitecting: Where to Begin

Optimizing application and system code for a new platform can be a daunting task. Briefly, you should consider three approaches or focus areas:

Static Analysis and Team Experience

Your organization probably already employs some form of static analysis tools and disciplines. Your team also possesses a wealth of a priori knowledge about and real-world experience with the code undergoing migration. Using this mix of tools and talent, begin by reviewing:

- Legacy main-line / main-loop
- Identified most-called functions implemented by your application (top 15%)
- Complete inventory of most frequently and least frequently-called RTOS APIs
- Known critical paths and bottlenecks

and by examining:

- Mapping of RTOS APIs onto Linux repertoire (See next section.)
- Shared data structures
- Use of IPCs and synchronization mechanisms

Just this level of analysis will highlight your primary candidates for re-architecting.

Dynamic Analysis

Use of dynamic analysis will confirm raw static frequency analysis and provide guidance on where to spend your engineering budget in optimizing and re-architecting.

A key exercise is to compare where your legacy application spent its time in its original hosting vs. time spent after migration. An important metric is the ration between time spent in user code vs. in system libraries and kernel execution. MontaVista DevRocket features a number of capabilities in this area.

Real-Time and Run-Time Performance Analysis

One of the first areas your team is likely to examine is performance. Linux may very well meet your legacy performance requirements, or there may exist performance gaps to be closed. In any case, performance is a good candidate for tuning and re-architecting. Metrics of merit include

- Interrupt latency
- Preemption/scheduling latency
- Start-up/boot time

Again, MontaVista DevRocket provides valuable tools and capabilities to ease this kind of evaluation.

APIs (Applications Programming Interfaces)

While the benefits of moving to Linux 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.

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()	
t_resume()	q_send()			
t_setpri()	q_urgent()			
t_setreg()	q_vcreate()			
t_start()	q_vdelete()			
t_suspend()	q_vident()			
	q_vreceive()			
	q_vsend()			
	q_vurgent()			
	q_vbroadcast()			
	ev_receive()			
	ev_send()			

Figure 6 –Typical frequently-used pSOS APIs

Vendors used to characterize and even promote their kernels in terms of the number and type of systems calls. For performance reasons, RTOS system call were often implemented by direct subroutine calls instead of by traps or exceptions used in real protected OSes like Linux. In practice, application code accesses to system calls occur through libraries that either act as wrappers for the system calls or even implement entire functions inside the library code without ever calling the kernel. Examples of the first case are task creation and scheduling calls, like VxWorks taskInit(); 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 7 –Typical frequently-used VxWorks APIs

Your application probably makes no distinction between direct 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 RTOSs have accrued hundreds, even thousands, of APIs in their decades of commercial existence and it is not practical to address the mass of those APIs. 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 6 and 7.)

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.

Many commercial RTOSes include standard call sets from POSIX or BSD, but those APIs often exist only as window dressing. 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 is prevalent in UNIX-based open systems and in government and military arenas. However, POSIX 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 (US-based National Institute of Standards and Technology), but now falls under the auspices of the IEEE as IEEE1003 and other standards. In the last decade, POSIX has undergone several revisions, mostly recently to POSIX 2000.

Two important notions exist with regard to POSIX: compliance and conformance. Compliance implies that a given OS platform implements some subset of the standard and that the implementation is documented. Even platforms implementing a trivial subset can be termed POSIX compliant. Conformance, conversely, presents much stricter criteria, meaning that an OS has been subject to a certification test and passed within allowable parameters.

POSIX family 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. It is divided into four parts:

- Part 1: Base Definitions
- Part 2: System Interfaces
- Part 3: Shell and Utilities
- Part 4: Rationale

Embedded Linux, as an instance of UNIX (a UNIX work-alike), is POSIX-compliant operating system. Although conformance is technically an all-or-nothing metric, versions of embedded Linux anecdotally exhibit ever-increasing levels of conformance (90% or greater) against NIST and also Open POSIX test suites, depending upon architecture. Few if any RTOSes approach even 10% of POSIX.1, but some can implement subsets of POSIX.1b and .1c.

POSIX.1b

Originally part of POSIX “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 include POSIX.1b subset interfaces as alternatives to their native IPC/synchronization APIs. Linux in general track POSIX.1b, but not religiously, implementing and encouraging use instead of the more common SVR4 versions of the same mechanisms. An important POSIX.1b call in Linux that does not exist in RTOSes is mmap(), key for mapping memory-mapped devices into logical address space (and into device drivers).

POSIX.1c7

POSIX.1c details the APIs and semantics for POSIX threads, or pthreads, whose definition is somewhat intertwined with notions from POSIX.1b. Linux implements pthreads as a standard user level library, NPTL (New POSIX Threads Library). NPTL provides application threading services, wrapping around the native Linux kernel threading interface (clone() et al.). The pthreads interface is the closest analogue for RTOS tasks when porting code to Linux.

POSIX.13 and POSIX Profiles

POSIX.13 comprises derivative profiles for sub-setting POSIX.1 standards. It is germane in that the Embedded Linux Consortium considered it for inclusion in its own standardization efforts Platform Specification and that many RTOSs cite POSIX.13 compliance as well. The POSIX.13 profiles primarily exist to allow non-UNIX/Linux platforms to masquerade as POSIX compliant. In general, Linux satisfies PSE 50-54 requirements by being a superset of all of them, inclusively. Attempts to trim Linux to implement a PSE subset exclusively, amount to “radical codectemy” and create an OS that is no longer Linux nor a UNIX/POSIX OS.

SVR4, BSD, and Other UNIX APIs

SVR4 (UNIX System V, Revision 4) and versions of the Berkeley Software Design-derived BSD UNIX 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 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).

The glibc run-time can present memory footprint challenges to embedded applications, migrated or otherwise. For size-sensitive applications, MontaVista also offers µClibc (micro-C libc).

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 sleep() and nanosleep()
Watchdogs, task regs, partitions/buffers	Emulated by tool kits

Figure 8. – RTOS and Linux IPC and synchronization mechanisms

IPCs and Synchronization

Every operating system, whether general-purpose or embedded, supports 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 RTOS-to-Linux mapping is seldom completely one-to-one and that even when apparent IPC equivalents exist, their scope may be focused on communications among 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 8.

Mechanisms

Here 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 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 interfaces offer mutex/condition variable-based synchronization and mutual exclusion.

Queues and Mailboxes

Most RTOSs 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 and POSIX.1b mqueues.

```
#include <sys/mman.h>
#define REG_SIZE   0x4          /* device register size */
#define REG_OFFSET 0xFA400000   /* physical address of device */

void *mem_ptr;                  /* de-ref for memory-mapped access */
int fd;                        /* file descriptor */

fd=open("/dev/mem",O_RDWR);    /* open phys memory (must be root) */

mem_ptr = mmap((void *)0x0, REG_SIZE, PROT_READ+PROT_WRITE,
        MAP_SHARED, fd, REG_OFFSET);
                               /* actual call to mmap */
```

Figure 9. – Using POSIX mmap() to access memory-mapped peripherals

Shared Memory

Most RTOS-based programs make use of informally shared data structures. Some RTOSs 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 RTOSs 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, ATCA or other interconnects to be mapped into Linux virtual address spaces.

Events and RTOS Signals

While some RTOSs implement events as message-bearing queues, on most systems 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 pthreads.

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 strives to use “real” time (seconds, microseconds, or nanoseconds).

The Linux 2.6 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 therefore seldom implement down to the putative resolution of their parameters (e.g., nanoseconds). Your application should also account for jitter, especially for resolutions approaching system clock ticks (jiffies).

Periodic Task Execution

While most RTOSs 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

With earlier implementations of Linux, the OS offered the above timer and delay interfaces, but the clock resolution of those mechanisms was extremely coarse. The 2.6 Linux kernel supports high resolution timing as a standard feature, in theory limited only by the granularity of the system clock. You can determine the system clock resolution with the call clock_getres().

Watchdog Timers

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

While Linux offers other means to increase system robustness, applications may still need watchdog-like functionality. Timer signal handlers can easily emulate such terminator behavior, and standard watchdog code also exists for many board-level configurations.

API and IPC Accommodation Strategies

We have looked at common calls and IPCs for VxWorks and pSOS. Other commercial or in-house RTOSs are likely to implement comparable calls, but are just as likely to feature their own unique APIs and IPCs.

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

1. Equivalence

Many RTOSs offer calls completely or nearly identical to Linux APIs. Because many RTOSs 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.

2. 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 methods.

Emulation can carry performance costs, and developers always assume that emulated code runs less efficiently than the original native construct. Anecdotally, many applications have actually experienced performance increases, both in general performance and in the area of networking. Your mileage may vary!

3. 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 convenient.

Migration Benefits: Improved Reliability

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, 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. 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 or software induced panics.

Most often when a program goes awry, it does so silently, so 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 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 its original cause.

Built-in Reliability from 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.

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 limited in scope to an individual process, which receives the “segmentation violation” signal (SIGSEGV) when the fault occurs. These errors include

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

By default, the programs receiving a SIGSEGV terminate, but signal 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, which staunches memory leaks and permitsthe clean restart of the failed process without need to reboot the system. Moreover, “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.

What about RTOSs with Memory Protection?

The majority of this white paper discusses classic RTOS architecture: “flat,” physically addressed memory, little or no use of MMU (other than to configure cache), with all the deficits described above. However, a handful of RTOSs do leverage on-chip memory management, and offer protection that falls into three categories:

1. Static Partitioning

Several RTOS suppliers have over the years offered the capability to set up static partitions or protection blocks in user application and system software. Configured at boot time, these partitioning schemes tagged program code and constant data as read-only, causing exceptions on attempts to write to addresses in those partitions. Such schemes had no ability to protect system data, user data, and stacks (lacking lengthy context-switch hooks), so they were primarily useful for debugging. When an exception did occur, the OSs in question offered no fault resolution other than rebooting. However, restarting an errant task or continuing to run after a fault risked additional faults from latent corruption or exhaustion of dynamic resources (memory leaks, TCB overruns, etc.).

Migrating from statically partitioned systems presents mostly the same challenges (and benefits) as moving from a classic legacy RTOS. Information emitted by linkers to inform RTOS run-time code can usually be discarded or taken over transparently by the Linux loader. Explicit calls to protect code and data by user applications can usually be stubbed out since Linux by definition protects those same critical resources and more.

2. Dynamic Partitioning and Basic Process Model

A handful of classic RTOS suppliers took additional steps to create dynamic protection schemes that accommodated the memory maps of individual tasks, protecting both code and data of background tasks from corruption by executing application code. In some cases, these RTOSs approximated something resembling a POSIX process model, although usually without ability to recover process resources (as Linux does). Often these schemes protected waiting tasks from the foreground task but left the OS itself exposed; running code needed to interact with the OS but could likewise easily corrupt it. Examples of this kind of OS include VxWorks AE, VxWorks 6.0 and Integrity.

As with RTOSs offering static partitioning, migration to Linux from theses OSs is mostly transparent and the benefits of migration are compelling. Developers will want to stub out or remove legacy partitioning calls and convert legacy shared memory interfaces to those in Linux.

3. Full Virtual Addressing: Embedded UNIX

A small and different set of legacy embedded OSs offer many or most of the benefits of UNIX/POSIX process model present in Linux. They implement dynamic memory protection and shared memory in the same fashion, using the same APIs, usually with paged memory. Examples of this kind of system include LynxOS, QNX, and ChorusOS.

While comparable in terms of overall robustness and resource management, these systems present their own challenges to developers of next-generation applications:

- Proprietary, closed source code
- Limited availability of board support and device drivers
- High run-time licensing costs
- Small user communities and tiny developer populations
- Limited interoperability with enterprise and desktop implementations of Linux, UNIX and Windows
- Poor scalability, especially for very large and very small memory sizes, and for multi-core and multi-processor applications

They also can suffer from subtle technical issues. One such RTOS exposes/maps shared resources (e.g., all semaphores) to all user processes, risking system corruption; another uses does not truly support paged memory mapping, limiting run-time response to heavy loading.

Migrating from these legacy UNIX-type RTOSs to Linux is mostly transparent. These systems strive to be POSIX compliant and in some cases conformant, simplifying porting of application code and some device driver code as well. In some rare cases, legacy code will use POSIX APIs that do not yet exist in Linux, and will require mapping to equivalent SVR4 or BSD APIs.

Application-Specific Migration

Each embedded application presents its own design and performance challenges, in its original form and when retargeted to embedded Linux. Most of this white paper has treated the application space as universal, targeting all versions of MontaVista Linux, starting with Professional Edition. However, applications in telecommunications infrastructure and in mobile telephony can benefit from capabilities specific to MontaVista Linux Carrier Grade Edition and MontaVista Mobilinux, and to the partner ecosystems around them.

Porting to MontaVista Linux Carrier Grade Edition

The Carrier Grade Linux (CGL) requirements specification created by members of the Open Source Development Lab (now the Linux Foundation) provides a framework, APIs, and resources for building standards-compliant highly available applications. Developers migrating legacy applications to MontaVista Linux Carrier Grade Edition (CGE), a registered and compliant CGL implementation, can take several paths towards creating more reliable and available applications:

Generic Legacy Applications

Legacy applications that do not involve specific fault resilient or availability-enhancing capabilities can benefit in several ways from migration to CGE.

Improved overall reliability: Just rehosting on CGE imbues legacy applications with greater fault resilience through standards-based fault isolation, management, and resolution in the CGL platform.

Better system management: Legacy code can be enhanced to use SAF HPI and other CGL-required capabilities to manage next generation hardware platforms like ATCA and BladeCenter.

Adding HA Wrapper Code Using HA middleware solutions from MontaVista partners like OpenClovis and GoAhead, developers can easily create “wrappers” for general purpose code needing additional reliability/ availability. Such wrapper code represents a low investment, low risk path to enhancing up-time and reducing failover latency through software and hardware management.

Full CGL and Middleware Port: Developers can realize the maximum benefit, incrementally or in one fell swoop, by migrating and rearchitecting generic legacy code to a highly available platform based on MontaVista CGE and supporting middleware.

Fault-Resilient Legacy Code

When legacy code includes its own fault-resilience or high-availability scheme, migration can present fewer challenges, and in some cases, new ones. If the legacy code employs standards-based APIs and management architectures (e.g., SAF HPI or commercial HA middleware), then moving to CGE will not differ greatly from migrating non-HA applications. Legacy highly-available functions and semantics will be comparable or in some cases identical to those present in a CGL stack. If, however, the legacy fault resilience and management scheme is highly proprietary and application-specific, then your mileage will vary.

See the following section, Leveraging the MontaVista Partner Ecosystem, for information on HA software solution suppliers who can make this transition easier.

Porting to MontaVista Mobilinux

OEMs have deployed Linux on dozens of new mobile handsets and other portable devices. While most of these deployments are for new designs, most bring with them a range of legacy technical requirements and also legacy source code for graphical user interfaces (GUI), personal information management (PIM), telephony API (TAPI), baseband and multimedia. A few of these domains merit specific consideration.

GUI and PIM: Legacy mobile application code usually builds on standard user interface frameworks (like Qt or Gnome), on Java, on in-house UI frameworks, or on combinations of the three. Porting standard UI code is mostly a matter of rebuilding legacy code for the Linux-based environment and/or new CPU. With Java-based application code, no rebuilding of application at all is required, as long as the legacy and target environments employ the same Java profile (e.g,. J2ME) and the same Java graphical toolkits (awt, Swing, etc.). Accommodating changes in display size, color depth, and input methods can require additional engineering investment.

TAPI and baseband: Today, OEMs primarily deploy Linux on higher-end smartphones and well-provisioned feature phones. On these devices, the baseband interface is encapsulated within a GSM or CDMA “modem” addressed by the applications CPU via a serial interface, on top of which runs a TAPI library. Each of the major handset manufacturers has their own TAPI, although some common standards are emerging. As mid-tier and low-end phones begin to deploy Linux, we will see increasing numbers of “single core” implementation with baseband and applications running on a single processor, requiring both migration of legacy baseband code to Linux and real-time response to meet both legacy and emerging timing requirements for GPRS, CDMA, and other wireless standards.

One path for legacy mobile migration to Linux lies in virtualization and partitioning. By insinuating a virtual machine layer between mobile hardware and system software, embedded virtualization allows Linux to run in one (or more) partitions while an entire legacy RTOS and stack (typically based on Nucleus) run in another. This approach allows extremely rapid migration to new mobile designs by preserving existing legacy functionality in place (in its own partition), supporting innovative and differentiating Linux-based applications to run alongside legacy code. Virtualization provides a strong option for migrating all types of legacy mobile code, from GUI to applications to baseband to multimedia. To learn more about options for virtualization and partitioning, see Leveraging the MontaVista Partner Ecosystem below.

As with a range of other embedded applications, a full native port is still the best way to gain maximum benefit from migration. MontaVista Mobilinux offers developers a range of native capabilities to meet both legacy and emerging demands of mobile telephony: power management, footprint scaling, flash and RAM file systems, and other technologies.

Other Issues

This white paper focuses on APIs, IPCs, and porting architectures. Other interesting RTOS migration issues not targeted here (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.)

Leveraging the MontaVista Partner Ecosystem

A wide range of resources exists in the MontaVista Partner Ecosystem, from tools to middleware to services. Following are listed just a few with direct application to the topic of this white paper:

RTOS API Emulation

MapuSoft

MapuSoft offers its OS Changer compatibility API libraries to support legacy pSOS, Nucleus and VxWorks applications running on the MontaVista Linux platform. OS Changer supports/emulates the most common APIs from these legacy systems, speeding migration for lab use or even deployment.

To learn more, visit http://mapusoft.com

Virtualization Platforms

VirtualLogix

VirtualLogix provides VLX real-time virtualization technology, which enables multiple “guest” operating aystems to run simultaneously on the same single-core or multi-core processor. VLX lets both legacy RTOS applications and new value-added applications based on MontaVista Linux run together in deployed systems, easing migration by preserving legacy source code and performance attributes intact. VLX run-time virtualization offers benefits to mobile, telecoms infrastructure, and a range of other application types. For mobile, it lets device OEMs continue to deploy RTOS-based base-band, multimedia and other stacks on a single processor core alongside new Mobilinux applications, each in separate and secure partitions. For carrier grade applications, VLX lets developers refactor and consolidate legacy line-card applications into scalable blade-based systems running on single or multi-core CPUs or across multiple blades.

To learn more, visit http://virtuallogix.com

VMware

VMware Workstation supports migration by providing a sandbox for legacy build environments and also by providing a virtual target for prototype execution in the absence of actual target hardware. MontaVista Application Developer Kit (ADK) includes a copy of VMware for to support prototyping and debugging of CPU-independent code.

To learn more about VMware Workstation, visit http://www.vmware.com

Carrier Grade Linux High Availability Middleware

GoAhead Software

GoAhead Software supplies embedded developers with an end-to-end high availability and systems management solution, offering platform independence and support for leading hardware types, (including ATCA, cPCI, BladeCenter, MicroTCA, etc.). GoAhead SelfReliant also supports Service Availability Forum (SAF) Hardware Platform Interface (HPI) specification for complete chassis and blade server management of hardware-specific sensors and controls.

GoAhead SelfReliant eases migration of legacy applications to MontaVista Linux by providng standards-based HA middleware on both Linux and a range of RTOS platforms. Learn more at http://goahead.com

OpenClovis

The OpenClovis Application Service Platform offers developers a low-risk path for migrating legacy fault-resilient applications to a modern carrier grade Linux platform. OpenClovis provides wrappers for legacy software lacking any “HA awareness”, standards-compliant SAF APIs for resource management, and advanced APIs for program life-cycle and system monitoring/management. With OpenClovis, developers can overcome migration challenges for highly available applications on MontaVista Linux Carrier Grade Edition (CGE) by preserving existing code, by mapping legacy management software onto modern standards-based M/W, or by supporting complete migration to OpenClovis.

Learn more by visiting http://www.openclovis.com

Development Tools

KlocWork

Klocwork K7 performs comprehensive static analysis for C, C++, and Java applications. K7 helps developers address a range of quality issues including defects and vulnerabilities, architecture and header file anomalies, while also providing software metrics in a range of reporting forms and formats. Many developers find K7 useful in the migration process for helping to document and understand legacy software architecture, API usage and calling chains.

Learn more about KlocWork K7 at http://www.klocwork.com

Other Partners

More than 250 partner companies host their offerings on MontaVista Linux platforms. To learn how these complimentary technologies can help you to migrate legacy and create new applications, visit http://www.mvista.com/partner_portal.php

Conclusion

The move is on. Developers are leaving behind first generation RTOSs 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

Gallmeister, Bill. POSIX.4 Programming for the Real World. (Sebastopol, Calif.: O'Reilly) 1995.

Haraszti, Zsolt. Migrating Legacy Applications to COTS High Availability Middleware, (Petaulma, Calif.: OpenClovis), 2006.

Linux Foundation. Carrier Grade Linux Specifications, versions 3.2 and 4.0, 2007.

Montalban, Manuel. "Can Virtualization Pave the Way to Embedded Open Source Advantage?" Presentation at Informa Open Source in Mobile, (Amsterdam), 2006.

Nichols, Buttlar, and Farrell. Pthreads Programming: A POSIX Standard for Better Multiprocessing. (Sebastopol, Calif.: O'Reilly), 1996.

Open Group. The Open Group Base Specifications Issue 6 IEEE Std 1003.1, 2004.

Weinberg, William. "Porting RTOS Device Drivers to Embedded Linux," Linux Journal n. 126: October 2004.

Weinberg, William. "Migration from UNIX to Linux". Presentation at LinuxWorld Expo (San Francisco) 2005.

Weinberg, William. "Moving from a Proprietary RTOS to Embedded Linux." RTC Magazine, April 2002.

About MontaVista Software

MontaVista Software, Inc. is the leading provider of Linux for intelligent devices and telecommunications infrastructure. MontaVista delivers a commercial-quality Linux operating system, time-saving development tools, expert support, and design and migration services. MontaVista has more than 400 partners, and thousands of companies use MontaVista embedded Linux to add functionality, increase reliability, reduce costs, and accelerate product development. MontaVista has offices in 15 countries. For more information, please visit www.mvista.com.

montavista™

MontaVista Software, Inc.
2929 Patrick Henry Drive
Santa Clara, CA 95054

Tel : 408.572.8000
Fax : 408.572.8005
email: sales@mvista.com
www.mvista.com