Migrating Legacy VxWorks Applications to Linux

Version 3.0.1 – September 17, 2008

Prepared by

Bill Weinberg, Linux Pundit for MontaVista Software
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>3</td>
</tr>
<tr>
<td>WHO SHOULD READ THIS WHITE PAPER?</td>
<td>3</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>4</td>
</tr>
<tr>
<td>MIGRATION ARCHITECTURES</td>
<td>4</td>
</tr>
<tr>
<td>Mapping Legacy Constructs onto Linux</td>
<td>6</td>
</tr>
<tr>
<td>THE MIGRATION PROCESS</td>
<td>9</td>
</tr>
<tr>
<td>Porting Considerations</td>
<td>9</td>
</tr>
<tr>
<td>Basic Steps</td>
<td>10</td>
</tr>
<tr>
<td>Rearchitecting – Where to Begin</td>
<td>11</td>
</tr>
<tr>
<td>APIS – APPLICATIONS PROGRAMMING INTERFACES</td>
<td>11</td>
</tr>
<tr>
<td>Key Standard APIs and Libraries</td>
<td>12</td>
</tr>
<tr>
<td>IPCS AND SYNCHRONIZATION</td>
<td>14</td>
</tr>
<tr>
<td>Mechanisms</td>
<td>15</td>
</tr>
<tr>
<td>API and IPC Accommodation Strategies</td>
<td>17</td>
</tr>
<tr>
<td>MIGRATION BENEFITS</td>
<td>18</td>
</tr>
<tr>
<td>The VxWorks Model – Maximum Exposure</td>
<td>18</td>
</tr>
<tr>
<td>Built-in Reliability from the Linux Programming Model</td>
<td>18</td>
</tr>
<tr>
<td>What about VxWorks Version 6.x and VxWorks AE?</td>
<td>19</td>
</tr>
<tr>
<td>FAQ - MISCONCEPTIONS ABOUT EMBEDDED LINUX</td>
<td>20</td>
</tr>
<tr>
<td>APPLICATION-SPECIFIC MIGRATION</td>
<td>22</td>
</tr>
<tr>
<td>Migration to Carrier Grade Linux</td>
<td>22</td>
</tr>
<tr>
<td>Migration to Mobile Linux</td>
<td>23</td>
</tr>
<tr>
<td>MIGRATION RESOURCES</td>
<td>23</td>
</tr>
<tr>
<td>Documentation and Training</td>
<td>24</td>
</tr>
<tr>
<td>DIY vs. Outsourcing</td>
<td>24</td>
</tr>
<tr>
<td>Open Source Projects and Commercial Products for Migration</td>
<td>25</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>25</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>25</td>
</tr>
</tbody>
</table>
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 VxWorks applications to run on Linux, this white paper will help you understand the transition process, to assess the challenges and risks involved, and appreciate benefits realized from such a move.

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

Who Should Read This White Paper?

This document is targeted to help technical decision makers assess the viability of migration from VxWorks to Linux. In particular, this document strives to assist

- Developers and Development Team Managers
- CTOs, VPs and Directors of Engineering
- OEM Product Management Teams
Migrating Legacy VxWorks Applications to Linux

Introduction

For the past eight years, Linux has unseated VxWorks as the embedded OS of choice for a wide swatch of embedded applications. Once considered to be a marginal and experimental platform, today embedded Linux is 100% mainstream and leads market and design share across key application areas, including mobile/wireless, data networking, telecommunications infrastructure, and consumer electronics. Indeed, industry analysts show embedded Linux garnering up to one third of 32 and 64 bit designs, well ahead of all other embedded OSes.

Notwithstanding the hard-won leadership position enjoyed by Linux in embedded, vast quantities of legacy designs, as well as a percentage of new applications, continue to build on VxWorks and similar RTOS software. While not every VxWorks-based design is a candidate for migration to a modern OS like Linux, many projects remain locked into the legacy Wind River OS due to

- Concerns about retooling / retraining for a new platform
- Misconceptions about Linux architecture, capabilities and performance
- Budget constraints for migration engineering

The purpose of this white paper is to address these and other concerns, to make a clear case for migration, and to elucidate both technical and financial benefits conferred by moving.

You should find this document useful if you are planning a move to embedded Linux in the near future or even if your team is just considering the level of investment to convert existing applications to run on embedded Linux. This paper will help you understand the transition process, assess challenges and risks involved, and appreciate the benefits realized from migration.

Migration Architectures

Linux may increasingly find itself in the place of legacy OSes, but the architecture of the Linux OS is very different from those legacy systems. Moreover, there exist multiple means to host legacy RTOS applications on a POSIX-type OS like Linux. The following section lays out three approaches to migration, from conservative paths that preserve legacy attributes and architecture to more extensive revamping of code and application structure.

Let’s compare and contrast three migration and re-hosting paradigms for legacy software under Linux:

- RTOS API Emulation over Linux
- Run-time partitioning with virtualization
- Full native Linux application port

RTOS Emulation over Linux

For legacy VxWorks applications to host and execute on Linux, a Linux-hosted run-time 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

---

1 Venture Development Corporation (VDC) 2006 and 2007.
do. In many cases new code must intervene to emulate missing functionality. Even when analogous APIs exist, they can present parameters that differ in type and number.

Legacy RTOSes can implement hundreds of system calls and library APIs; VxWorks documentation\(^2\) describes over one thousand unique functions and subroutines. Real applications use only a few dozen unique RTOS APIs and call functions from standard C/C++ libraries for the rest of their (inter)operation.

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 like MapuSoft (http://www.mapusoft.com). There also exists an open source project called v2lin\(^3\) 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 or running as an application “over” another virtual platform, where a piece of system software (running on “bare metal” or as a hosted application) enables the execution of one or more “guest” OS instances. In enterprise computing, Linux-based virtualization technology is a mainstream feature of the data center, but virtualization also finds many applications on the desktop and in embedded systems.

Data center virtualization enables server consolidation, load-balancing, creation of 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.

Embedded virtualization entails partitioning of CPU, memory and other resources to host VxWorks or other RTOS and one or more guest “application” OSes, usually Linux, to run higher-level software. Virtualization can support migration by allowing an VxWorks application and VxWorks itself to run mostly intact in a new design, with Linux executing in its own partition. This arrangement (see Figure 3.) is useful when legacy code has dependencies on VxWorks APIs and on performance characteristics, e.g., real-time performance or

---

\(^2\) VxWorks Reference Manual 5.3.1

\(^3\) v2lin originated at MontaVista Software and is now maintained by Mike Kemelmakher and Andrew Skiba. The project was recently rearchitected to work with newer POSIX-compliant versions of glibc.
specific implementations of protocol stacks. Embedded virtualization as such represents a short and solid bridge from legacy code to new Linux-based designs, but that bridge exacts a toll – OEMs will continue to pay legacy RTOS run-time royalties and will also need to negotiate a commercial license from the VM supplier.

A wide range of options exists for virtualization, including the mainstream KVM (Kernel-based Virtualization Manager) and Xen. Embedded-specific paravirtualization solutions are available from a range of independent software vendors, including

- Open Kernel Labs – [http://www.ok-labs.com](http://www.ok-labs.com)
- Trango – [http://www.trango.com](http://www.trango.com)

**Application Retargeting for Native Linux Execution**

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, 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.

**Mapping Legacy Constructs onto Linux**

The above architecture descriptions readily suggest a straightforward architecture for porting VxWorks code to Linux: the entirety of VxWorks application code (minus kernel and libraries) migrates into a single Linux process; VxWorks tasks translate to Linux threads; the physical memory space maps 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 & Thread Creation**

Whether you use RTOS emulation kits for Wind River VxWorks or other RTOSes, 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.

![Diagram of VxWorks and Linux kernel mechanisms](image)

**Physical Memory**
- Task
- Task

**VxWorks Kernel**
- Memory-mapped Devices

**Logical Memory**
- Linux Thread
- Linux Thread
- Process

**Linux Kernel**
- Memory-mapped Devices

**Figure 4. – Mapping VxWorks tasks onto Linux threads**

In general, processes are “heavier” than threads because they carry more context with them. A Linux thread context (like an VxWorks task) consists primarily of a subset of CPU registers, a stack, a current program counter, and entries in the kernel’s data structures. A POSIX-style process adds a complete virtual address space to this definition. At a minimum, Linux 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. So, while VxWorks and other RTOSes offer dynamic task creation APIs, RTOS vendors discourage frivolous and frequent task creation. The migration process provides a good opportunity to audit task/thread inventory of legacy VxWorks 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, reliability and resource management.

**Creating New Processes - Fork**

Task and thread creation in both VxWorks 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 process 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 usually accomplished 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.
```c
#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 6. – Listing of unified parent / child code with fork() call

Forking involves several steps (simplified):

- Create new virtual address space
- Map TEXT pages into new space (no copying – image is shared)
- Copy DATA pages (actually occurs per page, on first write\(^4\))
- Create copies of all current file descriptors
- Create scheduler entry (with clone() )
- Assign new PID
- 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 pthread_create() 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:

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

---

\(^4\) Linux strives to be “lazy” and postpones many such operations until the resource or state is actually required.
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 like FPU, MMX, Altivec and ARM co-processor registers.

Design Criteria – Processes or Threads
While a simple port will typically map VxWorks 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 application-level 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 best trade off for your particular application.

The Migration Process
The process for porting from VxWorks to embedded Linux is little different from moving 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 APIs, and IPCs.

Porting Considerations
Most developers using off-the-shelf VxWorks development kits (Tornado, for legacy VxWorks) will 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 legacy VxWorks.
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, you may find useful tools from vendors like Klocwork (http://www.klocwork.com).

A detailed code and API audit will reveal several classes of mapping and equivalence among calls to VxWorks 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 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 same 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 VxWorks 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 VxWorks header files can introduce conflicts with native Linux headers)
5. Add #includes for Linux header files to your 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. Incrementally recode and re-architect to leverage native Linux constructs
Rearranging – 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 knowledge from 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 VxWorks APIs
- Known critical paths and bottlenecks

and examining

- Mapping of VxWorks APIs onto Linux repertoire (see next section)
- Shared data structures
- Use of IPCs and synchronization mechanisms

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

Dynamic Analysis
Using 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 invaluable 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 VxWorks 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 VxWorks calls and features that do not readily translate into the Linux model.

For performance reasons, VxWorks system calls were implemented by direct subroutine calls instead of 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.

<table>
<thead>
<tr>
<th>Task Management</th>
<th>Message Queues</th>
<th>Semaphores</th>
<th>Watchdogs</th>
</tr>
</thead>
<tbody>
<tr>
<td>taskSpawn()</td>
<td>taskSafe()</td>
<td>msgQCreate()</td>
<td>wdCancel()</td>
</tr>
<tr>
<td>taskInit()</td>
<td>taskUnsafe()</td>
<td>msgQDelete()</td>
<td>wdCreate()</td>
</tr>
<tr>
<td>taskActivate()</td>
<td>taskDelay()</td>
<td>msgQSend()</td>
<td>wdDelete()</td>
</tr>
<tr>
<td>taskDelete()</td>
<td>taskName()</td>
<td>msgQReceive()</td>
<td>wdStart()</td>
</tr>
<tr>
<td>taskDeleteForce()</td>
<td>taskNameToId()</td>
<td>msgQNumMsgs()</td>
<td></td>
</tr>
<tr>
<td>taskSuspend()</td>
<td>taskIdVerify()</td>
<td>semGive()</td>
<td></td>
</tr>
<tr>
<td>taskResume()</td>
<td>taskIdSelf()</td>
<td>semTake()</td>
<td></td>
</tr>
<tr>
<td>taskRestart()</td>
<td>taskIdDefault()</td>
<td>semFlush()</td>
<td></td>
</tr>
<tr>
<td>taskPrioritySet()</td>
<td>taskIsReady()</td>
<td>semDelete()</td>
<td></td>
</tr>
<tr>
<td>taskPriorityGet()</td>
<td>taskIdGet()</td>
<td>semBCreate()</td>
<td></td>
</tr>
<tr>
<td>taskLock()</td>
<td>taskLock()</td>
<td>semCCreate()</td>
<td></td>
</tr>
<tr>
<td>taskUnlock()</td>
<td>taskUnlock()</td>
<td>semMCreate()</td>
<td></td>
</tr>
<tr>
<td></td>
<td>taskIsListGet()</td>
<td>semMGiveForce()</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6 – 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 VxWorks or Linux. Kernels like VxWorks, (and pSOS, VRTX, and Nucleus) 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 (Figure 6.).

**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. VxWorks includes standard call sets from POSIX or BSD, but those APIs exist primarily as “window dressing”. Most developers use, and Wind River recommends the proprietary, closed VxWorks APIs, and it is calls that lock projects into long-term commitments to that platform.

If you are porting standards-based code, whether implemented on VxWorks (native and/or POSIX subset APIs), 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\(^5\) 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

\(^5\) Pronounced like *positive*, not like *posey*
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 sub-set 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 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.1c**
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.

---

7 POSIX.1b and .1c are subsumed into 1003.1-2003
8 Then called POSIX.4
9 And supercedes earlier implementations, e.g., Linux threads and NGPT
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 10 and that many RTOSes 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 PSE50-54 requirements by being a superset of all of them, inclusively. Attempts to trim Linux to implement a PSE subset, exclusively, amount to “radical codeectomy” 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 `ipcs()` 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 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 the VxWorks library implementation.

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).

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

Figure 7. – RTOS and Linux IPC and Synchronization Mechanisms

IPC 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

10 See [http://www.embedded-linux.org/platform.php3](http://www.embedded-linux.org/platform.php3). Note that the Embedded Linux Consortium is now defunct.
mapping to Linux 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 VxWorks tasks, with subtly differing semantics.

The following sections survey the most common IPCs in VxWorks and other RTOSes and how they map onto Linux analogues, as summarized in Figure 7.

**Mechanisms**

Following is a brief discussion of the mapping of legacy IPCs and synchronization mechanisms onto their Linux equivalents as implemented for C language programming. The focus here 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 VxWorks 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**

VxWorks offers lightweight queueing constructs 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.

```c
#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 8. – Using POSIX mmap() to access memory-mapped peripherals*

**Shared Memory**

Most VxWorks 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 VxWorks adds value with shared memory mechanisms (with 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 Signals**

While some RTOSes 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-queueing 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 VxWorks 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.¹¹

**Timers and Task Delays**

VxWorks 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**

VxWorks 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 VxWorks timer implementations is that legacy timer APIs usually quantify time in terms of 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**

Legacy task delay calls translate into a family of Linux 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). Your application should also account for jitter, especially for resolutions approaching system clock ticks (jiffies).

**Periodic Task Execution**

While most legacy applications 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

Legacy applications 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

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

Equivalence

In many cases, VxWorks offer calls completely or nearly identical to Linux APIs. Because the original VxWorks developers were also UNIX programmers, that RTOS features 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 VxWorks APIs, while not differing greatly, will require “massaging” with the insertion of library code to emulate additional or different functionality.

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!

Recoding

When VxWorks 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

The architecture of VxWorks-based applications has changed little since its inception, almost two decades ago. Despite huge advances in microprocessors and other aspects of hardware design, VxWorks applications are structured as a set of tasks (C functions, typically), statically linked to run-time libraries (including the Wind 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 VxWorks kernel itself.

The VxWorks Model – Maximum Exposure

The time-worn and familiar VxWorks 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 VxWorks

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 VxWorks run-time 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”: an errant task can corrupt data and code anywhere in the VxWorks 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.

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 uninitialized 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\(^\text{12}\) by the operating system, staunching memory leaks and permitting the clean restart of the failed process without need to reboot the system. Moreover, “silent” corruption that can occur in VxWorks 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 VxWorks Version 6.x and VxWorks AE?**

The majority of this white paper discusses classic architecture of VxWorks and other common legacy RTOSes: “flat”, physically addressed memory, little or no use of MMU (other than to configure cache), with all the deficits described above. However, transitional RTOSes like VxWorks AE and more recently VxWorks version 6.0 do leverage on-chip memory management, and offer protection that falls into three categories:

**Static Partitioning**

Several suppliers have 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) and were primarily useful for debugging. When an exception did occur, the OSes in question offered no fault resolution other than rebooting; 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 (VxWorks 5.0 and earlier), Information emitted by linkers to inform legacy 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.

**Dynamic Partitioning and Basic Process Model**

Wind River and other 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 RTOSes approximated something resembling a POSIX process model, although usually without ability to

\(^{12}\) Resources shared among processes (shared memory, IPCs, etc.) are not recovered until all sharing processes terminate. Moreover, robust mutexes enable recovery / unlocking of held mutexes when the holding process terminates without releasing the mutex.
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 RTOSes offering static partitioning, migration to Linux from these OSes is mostly transparent and the benefits of migration compelling. Developers will want to stub out or remove legacy partitioning calls and convert legacy shared memory interfaces to those in Linux.

FAQ - Misconceptions About Embedded Linux

The goal of this document is to help developers plan for migration from VxWorks to embedded Linux, mostly by introducing concepts key to that migration process. However, many developers begin reading this document faced with misconceptions and FUD regarding the migration process. Several common areas of concern are documented in the following section:

Linux Networking – Up to the Task?

Many VxWorks developers come to Linux with two sorts of concerns about Linux networking:

The first is suitability. Many OEMs found the TCP/IP stack included with VxWorks to be buggy and unreliable. Furthermore, since VxWorks is a proprietary, closed source OS, they could not readily improve it and so had to replace it wholesale. Unlike the legacy VxWorks BSD-based networking stack, Linux TCP/IP is completely open source – you can review the code, make changes and submit patches back to community repositories. Moreover, the Linux TCP/IP stack powers tens of millions of Linux-based web servers and ubiquitous routers, switches and internet appliances. This broad choice of both IT managers and OEMs is further supported by independent source code quality studies by companies like Coverity (http://www.coverity.com) and Reasoning (http://reasoning.com/)

The second area of concern is performance. Wind River and others legacy vendors have published synthetic benchmarks (usually with netbench), predictably showing Linux networking as deficient when compared to VxWorks and other legacy platforms. These benchmarks invariably involve arbitrarily small packet sizes (≤16 bytes), atypical of real-world applications. For packet sizes of 32 bytes and larger, Linux shows clearly higher throughput and continues to make gains as packet size increases. This performance advantage comes from better marshalling algorithms and more robust memory management, bolstered by a global community development model.

Can Performance-sensitive Code Reside in User Space?

With VxWorks, no distinction exists between system code and user code. All execution occurs in privileged mode; device drivers, interrupt handlers, time critical operations, and mainstream application code all run as if they were part of the kernel (albeit with different priorities). By contrast, Linux leverages native CPU mechanisms for segregating code into system-critical operation (kernel space) and user applications (user space). But Linux also employs a design philosophy of minimizing critical, kernel-level code and of avoiding system calls altogether whenever possible (e.g., implementing systems functions in user space libraries).

When migrating your embedded applications to Linux, many developers find themselves tempted to insert all critical code into kernel space. Certainly, Linux architecture predicates that the “top half” of devices drivers do reside in kernel space. But the “business end” of drives, as well as other I/O-related and real-time code need not migrate to kernel space as well. The privileges accorded to kernel space execution lie in the domain of security and access to kernel data, not to performance and throughput. Indeed, your project will be better served by migrating such code to
user space where it can be prioritized and tuned appropriately, not into the kernel. In kernel space, application-specific code can end up competing with mundane but critical kernel operations; it will lack access to standard libraries like glibc; it will bloat your system’s trusted computing base (TCB); and it can potentially destabilize system-wide performance.

What Happens to Process Resources at Exit?

In VxWorks, all running tasks allocate dynamic resources from a shared pool using APIs derived from and working similarly to the C library malloc() function. When a VxWorks task hangs or terminates abnormally, it is practically impossible to recover dynamic allocations – cooperative memory allocation leads to system-wide memory leaks and eventually the need to reboot.

As with VxWorks, Linux processes (and threads within them) can dynamically allocate resources as needed in the course of execution. However, upon termination, graceful or otherwise, the Linux run-time is able to recover those resources – even resources shared with other processes.

Why Does Linux Require a File System?

Legacy VxWorks systems initialize by loading a single image from ROM to RAM. That image contains the VxWorks kernel, libraries, drivers, applications and any other code and often data of interest. In this context, the notion of a file system entails additional software to access external storage media.

By contrast Linux requires the presence of a file system to operate. The main reason behind this requirement is that programs in Linux, starting at boot-time with init, are invoked by supplying a file descriptor / pathname for retrieving a program image. The Linux loader and APIs like execv() follow this paradigm, regardless of whether the program in question is a binary executable, a script or other program entity.

Linux does not require, however, the presence of rotating media, i.e., disks. Embedded designs and even desktop and server distributions can operate from file systems implemented in RAM, Flash, over a network, and/or with “disks” implemented as solid-state devices or on media with spindles and platters. Tens of thousands of Linux-based applications are deployed this way, with no impact on footprint or performance.

Must OEM Product Updates Be Delivered in Source?

With legacy VxWorks, updates, upgrades and patches usually require a complete rebuild of the image containing the VxWorks kernel, libraries, drivers, application code and data. OEMs, their channel and/or their customers must acquire a complete new software load, or portions thereof in source code, and build and reconstitute a system image. The target system must be shutdown and a BIOS or other firmware then mediates local or remote re-flashing of the main system ROM, followed by a reboot.

As noted in the previous section, all Linux systems deploy with some type of file system. While this requirement may seem burdensome (it is not), it also facilitates field updates and upgrades in either source or binary. Without interrupting operating applications, Linux-based devices can receive new versions of applications, libraries, drivers or other critical code and store them in the local file system (e.g., in flash). Individual applications can then restart with newly loaded pro-

---

13 Resources shared across processes are freed when the last sharing program terminates or frees them explicitly. See also Robust Mutexes - [http://devresources.linux-foundation.org/dev/robustmutexes/](http://devresources.linux-foundation.org/dev/robustmutexes/) and elsewhere.

14 At the process level. Threads within processes are launched comparably to VxWorks tasks, with pthread_create().
gram code without need for rebooting or other interruption. Patched libraries will be ready the next time user programs load or access them. And, modified device drivers can be stopped, unloaded and reloaded, in most cases without need for rebooting.

**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 Linux, starting with MontaVista Linux Professional Edition. However, applications in telecommunications infrastructure and in mobile telephony can benefit from capabilities specific to Carrier Grade Linux and Linux for Mobile applications, and to the partner ecosystems around them.

**Migration to Carrier Grade Linux**

The Carrier Grade Linux (CGL) requirements specification created by members of the Open Source Development Lab (now under the auspices of the Linux Foundation and of SCOPE) 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.
**Migration to Mobile Linux**

OEMs are deploying 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\(^\text{15}\) 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 and color depth, and input methods can require some additional engineering investment.

**TAPI and baseband** – today, OEMs are primarily deploying Linux on higher-end smart phones 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 along side it. Virtualization provides a strong option for migrating all types of legacy mobile code, from GUI to applications to baseband to multimedia.

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. Learn more at [http://www.mobilinux.com](http://www.mobilinux.com)

**Migration Resources**

Some readers of this white paper will find reassurance in the architectures, paths, examples and proof-points supplied to support and justify the choice to move off legacy VxWorks. Others may see the same information as a “glass half full” and be daunted by the number of choices and options for migration. The following section presents information of both technical and human resources for facilitating the migration process.

\(^{15}\) Very few, if any, legacy handset designs were based on VxWorks.
Documentation and Training

In stark contrast to legacy RTOS platforms like VxWorks, Linux source code and documentation is broadly available to a population that extends well beyond the cadres of embedded developers. Its global adoption for desktop, data center and of course device software mean that you and your team can look to off-the-shelf resources that include

- Readily available books on Linux programming, configuration, security, graphics, internals, drivers, etc. There are even a growing number of volumes that focus exclusively on embedded Linux, like *Embedded Linux Primer: A Practical Real-World Approach* (Prentice Hall Open Source Software Development Series) by Christopher Hallinan
- The Linux Documentation Project (http://tldp.org/)
- Myriad web sites documenting the same topics and more for enterprise, desktop and embedded applications
- Training from MontaVista and other Linux vendors as well as independent Linux training companies like K-Computing (http://www.kcomputing.com) and The PTR Group (http://www.theptrgroup.com)
- Information from semiconductor suppliers and chipset IP licensors (ARM, MIPS, et al.)
- Entire magazines and web sites dedicated to Linux-related subject matter, like *Linux Journal*, *Linux Magazine*, *Linux User and Developer*, and on the web LinuxDevices.com

This wealth of resources reflects the ubiquity of Linux knowledge, experience and expertise across the IT marketplace, supporting a significantly larger global talent pool than exists for VxWorks.

DIY vs. Outsourcing

The purpose of this white paper is to educate readers about accessible architectures and paths of least resistance to successful migration. The information presented is probably not sufficient to serve as a “how to” guide for DIY (Do It Yourself) migration, but the net message is not intended to discourage OEMs and developers from proceeding on their own.

Like many engineering projects, legacy RTOS migration boils down to some key Build vs. Buy decisions. To aid in this decision process, you and your team should ask yourselves

- What is your team’s pre-existing expertise in both VxWorks and Linux?
- How will training team members in either the source or target domain impact project schedules?
- What are the size and scope of the legacy code base and how transparent is that code to current team members?
- Is your team’s staffing sufficient to support the full legacy code base? To support the migrated code base, including Linux platform code and middleware?
- Does migration and related continuation engineering activities represent a core value-added activity or a marginal engineering investment?

The answers to these and other questions can help you decide whether to

- perform the migration yourselves or outsource the work to a Linux platform or services company
- integrate and deploy your own Linux platform from free software repositories or engage with a commercial embedded Linux supplier like MontaVista Software
**Open Source Projects and Commercial Products for Migration**

Throughout this white paper, the author has mentioned projects and suppliers whose wares simplify and accelerate the migration process. You should also review the members of the Monta-Vista Partners Program for additional suppliers.

**Conclusion**

The move is on – developers are leaving behind legacy VxWorks in search of more reliable and open embedded platforms like Linux. While the migration from VxWorks can 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.

**References**


HARASZTI, Zsolt [2006]. *Migrating Legacy Applications to COTS High Availability Middleware – Case Study by OpenClovis*.

LINUX FOUNDATION [2007]. *Carrier Grade Linux Specifications*, versions 3.2 and 4.0.

MONTALBAN, Manuel [2006]. “Can Virtualization Pave the Way to Embedded Open Source Advantage?” Presentation at Informa Open Source in Mobile, Amsterdam.


