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.

Legacy to Linux: Migrating from proprietary RTOS to Open Source Linux

Sachin Kaushik
Technical Marketing Specialist

MONTAVISTA SOFTWARE, LLC.

Contents

EXECUTIVE SUMMARY 3

INTRODUCTION 5

SOLUTION OVERVIEW 6

Migration Execution Architectures 6

The Porting Process 12

Migration Resources 16

CONCLUSIONS 18

APPENDICES 20

Executive Summary

Embedded applications migration from one version of system software to another is often a difficult task. Migration a real-time embedded application to a new OS is amongst the most challenging tasks. So, if you are considering the investment needed to convert your existing application to run on embedded Linux, this paper will help you understand the transition process, models available and some useful pointers.

(Note: The challenges and risks involved, along with various benefits realized by such move have been covered in a separate paper, Legacy to Linux: Challenges & Benefits).

At a very high-level, a simple architecture description suggests a straightforward architecture for porting RTOS code to Linux:

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

	Legacy OS	Linux
Process Model	No	Yes
MMU-enabled	No	Yes
User Space Addressing	Physical	Virtual
Kernel Space Addressing	Physical	Virtual
Driver Model	None/Partial	Complete
Interrupt Service	Anywhere	Driver Only

Fig a: Key Attributes of Legacy OS and Linux

This paper tries to highlight that Legacy to Linux Migration involves Real Investment,

- Usually entails multiple devices / interfaces
- Need to capture legacy code, technology, knowledge
- Modern programming model (Flat Memory to MMU plus)

However, such an approach presents your organization with,

Challenges

- Changes in design, practices, scope of code
- May need to deprecate s/w and h/w architectures
- Need to (re)train existing team, add new expertise

Opportunities

- Optimize platform, improve performance
- Unify fragmented internal platforms, code bases
- Create a more maintainable foundation for future
- Join with mainstream in embedded and enterprise

Introduction

Embedded system software and the open source Linux operating system have co-existed for a long time now. Companies using Linux for their embedded products find it time and cost efficient, when it comes to performance and maintainability. Another solution for embedded systems is a Real-Time Operating System (RTOS). The goal of this this paper is to investigate whether legacy RTOS based embedded design can be migrated to embedded Linux, 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.

While not every Legacy RTOS based design is a candidate for migration to a modern OS like Linux, many projects remain locked into the legacy Wind RTOS 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.

The Linux Operating System is used around the world to power all kinds of devices that require a full-featured OS, from heavy machinery to high-precision electronics. Also Linux development skills are commonly available from graduate student–level onwards in all degrees of expertise around the world. This makes it much less risky and more cost-competitive for product development than RTOSs.

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.

Solution Overview

This section will cover the challenges, approaches available with the developers today as well as process to achieve them. Lastly, it will highlight some of the perceived benefits and challenges when targeting "Legacy to Linux".

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. To emulate these interfaces for purposes of migration, developers only need a core subset of RTOS calls.

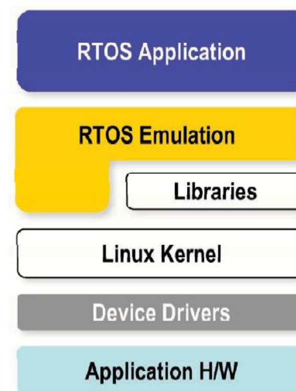

Figure 1. – RTOS emulation over Linux

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 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.

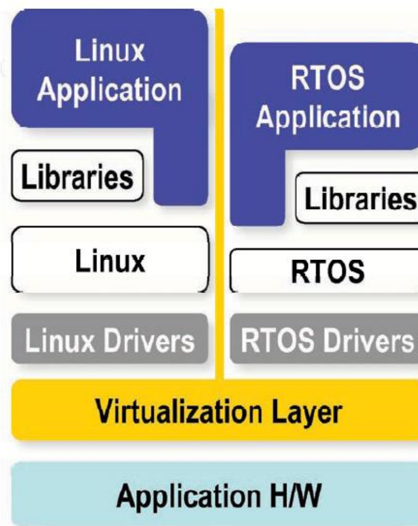

Figure 2. – Partitioned Run-time with Virtualization

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 2.) 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 toll OEMs 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 para-virtualization solutions are available from companies like VirtualLogix. (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.

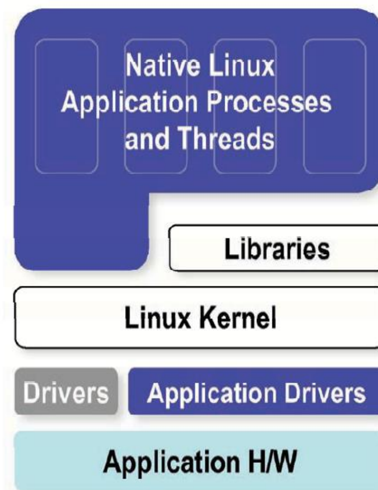

Figure 3. – Native Port of RTOS Application

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 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 eac type of entity, and performance and resource costs (and benefits) associated with each.

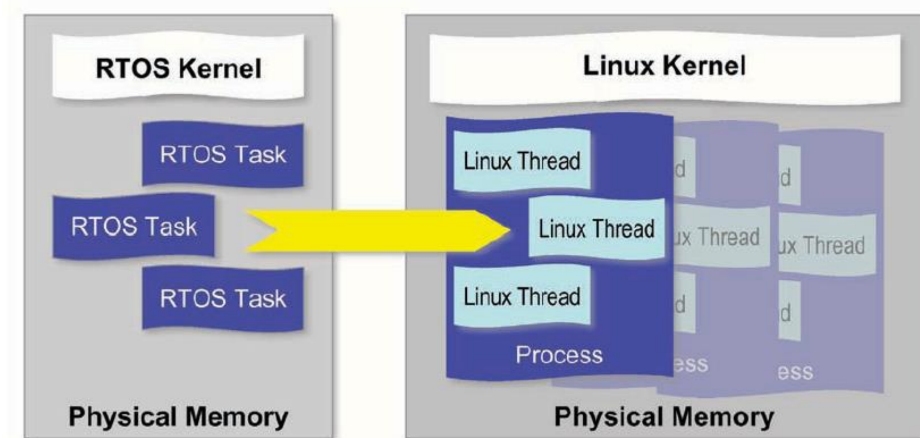

Figure 4. – Mapping RTOS tasks onto Linux threads

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 _P ID) {

 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.

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 Carrier Grade Edition (CGE) and/or Carrier Grade eXpress (CGX™) 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.

Re-architecting: 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.

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.

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 the following table:

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

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 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 RTOSs. 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.

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,
- The Linux Documentation Project (http://tldp.org/)
- Myriad web sites documenting the same topics and more for enterprise, desktop and embedded applications
- Yocto Project Docuementaion https://www.yoctoproject.org/documentation
- Training from MontaVista and other Linux vendors as well as independent Linux training
- Information from semiconductor suppliers and chipset IP licensors (ARM, MIPS, et al.)
- 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 Legacy RTOSs.

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. Buydecisions. To aid in this decision process, you and your team should ask yourselves

- What is your team's pre-existing expertise in both Legacy RTOS (e.g. VxWorks) and Linux?
- How will training team members in either the source or target domain impact project schedules?
- What is 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.

(Note: More information about the advantages and the engagement process of the embedded Linux professional services have been covered in a solution brief and same can be obtained from our external website www.mvista.com).

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 MontaVista Partners Program for additional suppliers.

Conclusions

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.

MontaVista has been an embedded Linux provider in the commercial space since 1999, and Linux has always been the only target area for our company, also the main idea the company was founded on was the ability to use Linux where traditionally RTOS-type OS:s have been used. Therefore we believe that we have unique expertise in helping our customers to migrate their existing SW investment over to Linux, taking advantage of the new HW and SW ecosystem and the advantages it provides in taking your products to the market faster, with more competitive features and less cost.

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.

This White Paper is for informational purposes only. MONTAVISTA MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS WHITE PAPER. MontaVista cannot be responsible for errors in typography or photography.

©2016 MontaVista Software, LLC. All rights reserved. Linux is a registered trademark of Linus Torvalds. MontaVista is a registered trademarks or registered trademarks of MontaVista Software, LLC. All other names mentioned are trademarks, registered trademarks or service marks of their respective companies

Information in this document is subject to change without notice.

Appendices

- William Weinberg, Moving Legacy Applications to Linux: RTOS Migration Revisited, 2014
- Iisko Lappalainen, Migrating from ITRON to Linux Concept White Paper, 2013
- 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.
- GALLMEISTER, Bill. O. [1995]. POSIX.4 : Programming for the Real World. O'Reilly & Associates; ISBN: 1565920740.
- HALLINAN, Christopher. [2006]. Embedded Linux Primer: A Practical Real-World Approach. Prentice Hall Open Source Software Development Series.
- WEINBERG, William [2005]. "Migration from UNIX to Linux". Presentation at LinuxWorld Expo, San Francisco - August 10.
- WEINBERG, William [2002]. Moving from a Proprietary RTOS to Embedded Linux. RTC Magazine, April.

MontaVista Software, LLC | 2315 North 1st Street San Jose, CA, 95131 | www.mvista.com

Doc Id: MVWP-LGC2LNX-051916