Migrating code to a latest version of the system software is a daunting task, even in the best case. Migrating from a legacy (usually closed source) Real Time OS to Linux™ is arguably much more difficult. ***Or is it?*** *This paper brings out the challenges and benefits of such a move!*

Legacy to Linux Migration: Challenges and Benefits

Sachin Kaushik
Technical Marketing Specialist

MONTAVISTA SOFTWARE, LLC.

Contents

Executive Summary

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

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

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

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

This white paper investigates whether a traditional proprietary RTOS can be substituted with embedded Linux, and if this kind of migration can lead to reduced licensing costs and increased general quality of the system.

Introduction

Embedded Linux, since it appearance in 1999, has unseated many legacy / proprietary / In-house Real Time OS (RTOS) like VxWorks and others as the embedded OS of choice for a wide variety of embedded applications. Once considered to be an experimental platform, today embedded Linux is 100% mainstream and leads market and design share across key application areas, including wireless, data networking, telecommunications infrastructure, and industrial electronics among others.

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 use Legacy RTOSs. While not every RTOS based design is a candidate for migration to a modern OS like Linux, many projects remain locked into the legacy 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 assess challenges and risks involved, and appreciate the benefits realized from migration.

Challenges with proprietary RTOSs today

In today's fast-moving and rapidly evolving world of software (SW) and hardware (HW) development, especially in embedded field, traditional RTOS specification has run into difficulties in providing the best baseline for competitive products.

This is largely influenced by the following major topics:

1. HW Ecosystem

The RTOSs have limited HW support, and lag behind when it comes to supporting newer SOCs. This contributes to a slower time to market and higher cost of development of products using RTOSs.

2. SW Ecosystem and Middleware support

There exists limited middleware and SW support for peripheral drivers available to RTOSs. Supporting new higher-level concepts using next generation buses such as USB 3.0 and newest networking features are hard or costly to implement. This contributes to either slower time to market and increased cost or lower product competitiveness.

3. Fragmented and local usage base

The RTOSs usage is almost exclusive to the small market segments (e.g. QNX for Automotive, ITRON for Japanese market etc.). This factor is largely behind the main issues 1 and 2 (as mentioned above) but will also mean that there is limited room for improvement to the situation and increased risk in taking on new product programs based on an individual RTOS.

4. Limited expert developer base

Since a single RTOS has a limited user base, there are limited developer resources available for working on product SW. This makes product development more expensive and resource scaling is made difficult.

5. Closed Source with Supplier Dependency

   Most proprietary RTOS are closed source i.e. no access to source code and vendor often have tight control over schedule and technology planning. This creates a huge dependency both in short term development planning and the long term maintenance of the embedded application.

Advantages of Linux

Linux on the other hand provides an alternative that has features that are largely the opposite of the problems with traditional RTOSs, namely:

1. HW, SW, Middleware Ecosystem and Vendor dependence

The Linux Ecosystem has, all things considered, likely the most advantageous baseline when it comes to developing SW on non-mainstream HW. For Linux, the latest SOC support is available from the launch of the devices, there is extensive Middleware support - often using an Open Source license - and the driver and pre-engineered application ecosystem available stands out in the top. Compared to RTOSs, this aspect will make it faster and cheaper to provide product SW, also typically with a more competent feature set. Also since Linux is ***always open***, no vendor lock-in is applicable here.

2. Global usage base and extensive developer community

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 to Linux, Challenges

The migration from an RTOS environment to Linux is certainly non-trivial and this section aims to give an outline of the major challenges related to the effort.

1. ***Fear of the un-known (Linux?):*** The first problem that comes up when gearing up for development in a new environment is the lack of initial expertise in the domain, as is the case with Linux as well. However, since the open nature of the Linux movement, such expertise is significantly easier to gather by exploring the materials produced by the ecosystem. Also by partnering with a commercial Linux vendor like MontaVista, further eases the transition and makes new competence building tools available, such as a dedicated training organization, customizable expert workshops and pre-made RTOS to Linux solution material.

2. ***Free Software (How?)***: Another non-technical topic often surfacing together with Linux is the open-source licensing, which can be seen as a risk by stakeholders from different communities. However, with expert guidance such as MontaVista can provide this risk can be in essence mitigated completely and Intellectual Property is not a concern with Linux.

3. ***GPOS vs RTOS (Is Linux Real-Time? Footprint? Boot-up time?)***: First and perhaps foremost of the technical challenges of migrating to Linux is the concern about Real-Time requirements and the capabilities of legacy RTOSs and multi-tiered operating systems such as Linux. While true that by default when using Linux version provided by the OSS community, the response latencies can be high and unacceptable for some RTOS-based designs, there are versions of Linux that are much more suited for applications with RT requirements.

Real-Time Requirements in Embedded Applications

All embedded applications have performance requirements. These include needs for rapid boot time, high networking throughput, graphical rendering speed, power management, “raw” computing power and other metrics of merit. Some applications, but not all, also have needs for real-time responsiveness. A good heuristic is

- 100% of embedded applications have a performance requirement.
- 40-50% of embedded applications have real-time performance requirements.
- 10-15% have some hard real-time requirements.
- < 5% have pervasive hard real-time requirements (all real-time, all the time).

The reality of this paradigm is backed up by studies that survey developer satisfaction with Linux real-time performance. For example, leading research firm's reports, several years running, those over 85% of developers are completely satisfied with Linux real-time response.

Approaches to Real-Time Linux

Real-time requirements come in more than one flavor – not all applications are created equal, nor are their needs for performance and responsiveness.

Segment	**Hard Real-time**	**Soft Real-time**	**Real-fast**
Telecommunications & network infrastructure	Frame-based protocols, Strict QoS, bandwidth reservation	Fault response, General QoS, VoIP	Most IP packet processing
Mobile communications & consumer electronics	RF baseband	Multimedia, networking	Imaging
Instrumentation and industrial control	Signal processing, data collection, motion control, robotics	Vision systems	Data analysis, CNC processing
Aerospace and defense	RADAR, SDR, surface control, target tracking	Display/imaging, security, intruder detection	Data analysis
Medical	Radiotherapy, MRI/CAT	Surgical assist, life support	Imaging

Figure1: Application segments and types of real-time requirements

Approaches to Real-Time Linux

Since the company was founded in 1999, MontaVista Software has invested substantial resources in enabling Linux to offer real-time responsiveness with native Linux constructs and APIs using open source software technology present in the main Linux source trees.

However, there exist a number of other paths toward real-time responsiveness in Linux that do not follow this open source, native, mainstream approach and a few merit description here:

Real-fast : using fast CPUs to meet deadlines
Fine tuning : tweaking parts of Linux to meet deadlines
Co-resident RTOSes : embedding a second OS for real-time
Virtualization : hosting a second (RT)OS on a hypervisor

	Co-Resident RT*	**Virtualization**	**Native Linux**
Real-time APIs	RT*-specific	Legacy RTOS APIs + hypercalls	Native POSIX threads and IPCs
Development environment	Additional SDK required	RTOS SDK + Hypervisor Tools (?)	Standard IDEs, e.g., devRocket
Performance	Hard Real-time	Hard Real-time	Hard Real-time
Limitations	▪ Minimal RT* Capabilities ▪ Requires special device drivers ▪ Tracking kernel revs	▪ Legacy RTOS capabilities ▪ Hypervisor IPCs ▪ Need to paravirtualize guest OS and drivers ▪ Limited CPU support	▪ Legacy RTOS capabilities ▪ Sub 50 microsecond worst case latencies
Source Code1	Partially Open	Closed	100% Open Source
Bill of Materials	Extra RAM/Flash for RT Components	Extra RAM/Flash for RTOS and Hypervisor	Zero Additional BoM Impact
Bill of Software	RTLinux Royalty	RTOS and Hypervisor Royalty	Royalty-Free

Figure 2: Comparing capabilities and impact of co-resident real-time with native Linux

By working in concert with the open source community at large and with key kernel maintainers in particular, MontaVista has been instrumental in developing and propagating numerous advances in the Linux kernel and user space libraries to enable native real-time. For real-time as well as for CPU and board support, development tools, power management and other embedded application enablers, MontaVista shows its ongoing commitment to community processes and open source software.

Today MontaVista Linux –based real-time solutions are controlling anti-aircraft guns, heart-rate monitoring equipment, industrial controllers and much more.

[1] At present, embedded hypervisors are not available as open source. It is theoretically possible to deploy Xen or KVM to achieve the same ends.

RT* Invented and patented by FSM Labs, and today the property of Wind River Systems, which rechristened it RTCore.

Now that we understand that Linux is a viable alternative to many embedded use cases. Let us look at some of the other challenges when considering Embedded Linux as a professional development environment for a time bound real world product development.

Challenges of Embedded Linux Development

Let's take a look at the primary challenges of embedded Linux development that every developer has to address at some point. They are:

- Assembling a software base
- Creating a development environment
- Keeping current with software changes

Additionally, the normal challenges of any embedded software development project or process must be addressed in the Linux environment as well:

- Configuration of the operating system platform
- Integration of custom applications
- Optimization for target hardware

Assembling a Software Base

One of the great strengths of open source software is the thriving and varied community of developers and software and the many alternatives this presents. However, a developer can invest many man-hours investigating the best solution for a particular development environment. In face a substantial amount of time can be spend merely understanding what the alternatives are. A developer on an embedded Linux project can waste a great deal of time simply surveying and selecting from the available components.

Commercial embedded developers may be more accustomed to proprietary products that provide a complete solution for a particular target board. In general, one does not find a complete solution in the open source community for embedded projects.

Simply bringing up a system for the first time requires selecting multiple components:

- A bootloader
- A kernel base
- A toolchain
- A basic application environment

It may be necessary to acquire these from multiple independent source locations. For example, MontaVista aggregates code from over 200 different open source projects for its distributions. Once downloaded, these components have to be ported to the target hardware, and then integrated with each other and maintained throughout the product life-cycle.

Some semiconductor vendors provide their own Linux distributions, which integrate a number of these items to provide a useful starting point. However, to fully leverage their hardware, these distributions often customize the kernel and other components in ways that are not compatible with other components developed in the broader community. Combining technologies from different sources may therefore present complicated integration issues down the road. This often makes it difficult to make use of important frameworks or applications from the open source community, and delays the start of actual product development.

In addition, semiconductor vendors generally do not provide support, maintenance or updates for their distributions. If they do, it's typically only for their largest customers, and then fairly limited

Creating a Development Environment

Most software development in the broader Linux community is based around "self-hosted development." That is, the host system (where development is done) and the target system (where the application will run) are the same. Because of this, little attention is often paid to a strict separation between the host and target environments. This frequently results in the application software having dependencies on the host environment. For embedded developers, "cross-development" is the rule, with significant differences between the host and target environments. The host and target may be running different operating systems, and often different processors. The target hardware, deliberately limited to meet costs, is often not capable of supporting the workstation-level storage, processing power and graphics capabilities that are desirable in a modern development system.

In order to create a complete development environment, an embedded Linux developer may have to assemble his own cross-development environment. Doing this includes the following tasks:

- Acquire a toolchain for the target hardware architecture
- Create a build environment around this toolchain
- Add the bootloader, kernel and base application software to this environment
- Eliminate host system dependencies from the application software
- Port the base software to the target hardware architecture
- Integrate debugging and analysis tools

The build environment needs to be flexible, in order to support the many disparate build frameworks that are in use in the open source community. The build environment also needs to insulate the product build from the host environment, so that a consistent product build can be produced on different host systems with reliable, repeatable results.

The base software selected must further be optimized for the target hardware and the target application. This task may include integrating or writing kernel drivers for the target hardware devices, as well as adding specialized open source or proprietary software applications and frameworks.

Keeping Current

Due to the rapid rate of innovation in the open source community, keeping up-to-date on changes to different software projects may involve following many disparate sources such as:

- Community mailing lists
- Vendor hardware mailing lists
- Security forums
- Web sites
- Source repositories

Software changes in the open source community are often distributed in the form of “patches,” which describe the source code differences from a previously released version of the software. If local changes have been made to the original community software, either by the developer or the vendor who provided the base code, there may be difficulties in applying the community changes to the current software.

Vendor and community distributions will also usually only fix bugs on the latest version of a distribution, and it can be a time-intensive task to back-port these fixes to a released embedded system that is based on an older version.

Sometimes bugs are addressed by systemic rather than narrowly targeted changes to the software, which not only makes back-porting more difficult, but also can introduce interface or behavioral changes that can require modification of other software in the system. Repeated thorough testing is necessary to ensure that the system continues to operate as intended, and that new issues are not introduced by the updates.

The integration of such changes is a continuous challenge not just during the development process, but also after release and over the lifetime of the product. Many members of the open source community may not be interested in helping with these issues, as they prefer to encourage developers to update to the latest version of the software. This tendency becomes stronger the older the base software version becomes, so products with long life spans in particular may require more direct expertise in the later years of their product life.

Dangers of These Challenges

Underestimating these challenges may lead to projects with ballooning costs, protracted schedule delays, or in a worst case scenario, both. For instance,

development of significant Linux expertise may be required to create a new distribution in-house. This often includes adding staff to focus on maintaining the open source software, not doing new development. This issue is frequently overlooked, as existing staff and management may underestimate the effort required to do this in a production environment, believing that open source software is an off-the-shelf solution.
The time required to develop such a project is also often miscalculated. Training of staff may be slower than expected due to poor, out-of-date, or non-existent documentation. Open source or semiconductor vendor code may be less robust than required for a production environment, or may not account for the limitations of an embedded environment, requiring additional development time and/or staff. Linux drivers may be unavailable for new or proprietary hardware, requiring additional development time, and perhaps highly skilled developers with specialized expertise.

Yocto Project and MontaVista: Simplify Open Source Software Development

Commercial embedded software development often has competing needs for higher flexibility with source control along with timely, cost-effective and stable releases. Using Yocto Project™, MontaVista® Linux, offers an ideal platform for developers who want to leverage the flexibility of a true open source development platform, as well as the ability to achieve rapid time to market.

MontaVista has been a forerunner and believer in bitbake and the OpenEmbedded embedded development paradigm. MontaVista took this to the next level by making our Carrier Grade Edition (CGE7) and Carrier Grade eXpress (CGX2.0) comply with Yocto 1.4 and 2.0 respectively. This allows MontaVista customers to take full advantage of the existing Yocto / OpenEmbedded ecosystem with its support for added feature layers and hardware support. Our next generation products will continue to align even more closure to this globally accepted standard. MontaVista as a Yocto Project participant member will continue to contribute to this open source collaboration for standardization.

MONTAVISTA AND YOCTO PROJECT

MEMBER OF ADVISORY BOARD WITH CODE CONTRIBUTIONS YOCTO LAYERS LIKE META-OPENEMBEDDED, OPENEMBEDDED-CORE, META-SELINUX, META-SECURITY& META-CGL AMONG OTHERS

ADOPTED YOCTO AS AN INTEGRAL PART OF OUR EMBEDDED LINUX DISTRIBUTION (PRODUCTS)

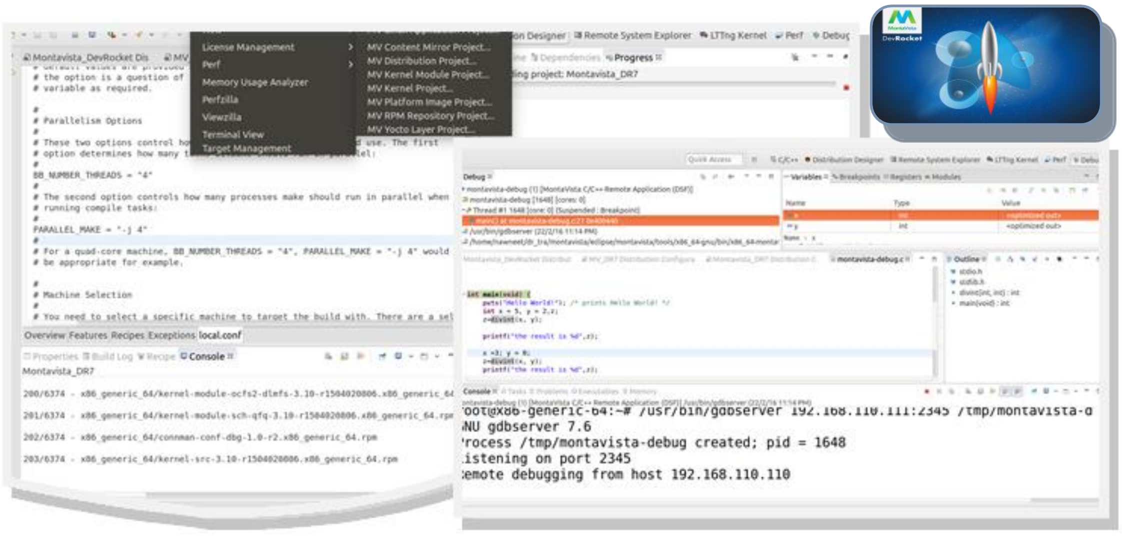
ADDING VALUE THROUGH COMMERCIALIZATION, SUPPORT, TRAINING AND MAINTENANCE OFFERINGS

yocto PROJECT
COMPATIBLE
MontaVista
Leveraging unique abilities of Yocto Project to deliver robust and scalable embedded Linux products

Yocto Project integration with MontaVista allows customers to fully customize their distribution using a familiar interface with easy access to source code, patches, and recipes on how to build binaries, while at the same time providing MontaVista binaries for those customers who don't have a need for customization. MontaVista compliments the distribution with custom developed features, leading edge development tools, technical support, and access to our world class engineering service.

MontaVista DevRocket™ Integrated Development Environment (IDE)

MontaVista DevRocket is the integrated development environment (IDE) that supports MontaVista Linux Platform and Application development. DevRocket delivers a set of tools designed to streamline and automate common embedded Linux development and analysis tasks, helping you deliver products to market faster. Based on standard Eclipse plug-ins, DevRocket significantly increases developer productivity by simplifying complex development tasks.


License Management
Perf
Memory Usage Analyzer
Perfzilla
Viewzilla
Terminal View
Target Management
MV Content Mirror Project...
MV Distribution Project...
MV Kernel Module Project...
MV Kernel Project...
MV Platform Image Project...
MV RPM Repository Project...
MV Yocto Layer Project...
Overview Features Recipes Exceptions local.conf
Console
Progress
Debug
Variables
Outline
DevRocket

```
# the option is a question of
# variable as required.

#
# Parallelism Options
#
# These two options control ho
# option determines how many t
#
BB_NUMBER_THREADS = "4"
#
# The second option controls how many processes make should run in parallel when
# running compile tasks:
#
PARALLEL_MAKE = "-j 4"
#
# For a quad-core machine, BB_NUMBER_THREADS = "4", PARALLEL_MAKE = "-j 4" would
# be appropriate for example.

#
# Machine Selection
#
# You need to select a specific machine to target the build with. There are a sel
```

```
Montavista_DR7

200/6374 - x86_generic_64/kernel-module-ocfs2-dlmfs-3.10-r1504020806.x86_generic_64
201/6374 - x86_generic_64/kernel-module-sch-qfq-3.10-r1504020806.x86_generic_64.rpm
202/6374 - x86_generic_64/connman-conf-dbg-1.0-r2.x86_generic_64.rpm
203/6374 - x86_generic_64/kernel-src-3.10-r1504020806.x86_generic_64.rpm
```

```
int main(void) {
    puts("Hello World!"); /* prints Hello World! */
    int x = 5, y = 2,z;
    z=divint(x, y);

    printf("the result is %d",z);

    x =3; y = 0;
    z=divint(x, y);
    printf("the result is %d",z);
```

```
'oot@x86-generic-64:~# /usr/bin/gdbserver 192.168.110.111:2345 /tmp/montavista-d
NU gdbserver 7.6
'rocess /tmp/montavista-debug created; pid = 1648
.istening on port 2345
:emote debugging from host 192.168.110.110
```

MontaVista DevRocket helps both the platform developer and application developer streamline the complex embedded Linux development process. By using the Eclipse-based DevRocket IDE developers can:

- Work in a familiar, standards based IDE alongside other Eclipse-based development tools.
- Easily incorporate internally developed or open source code in their projects.
- Perform detailed analysis of performance, memory usage, memory leaks at the platform and application level.
- Platform developers can easily download the source and build their target distribution.
- New images can be quickly and easily moved to the target board using the target management capabilities.

- Application developers can use the one click edit/compile/debug to quickly build and deploy target images.

Migration Benefits: Improved Reliability

The basic architecture of an RTOS-based application has changed little in the last 20 years, despite huge advances in microprocessors and other aspects of hardware design. RTOS applications are structured as a set of tasks (C functions, typically), statically linked to run-time libraries (including the RTOS kernel itself).

These tasks reside and execute in a single physical address space (in RAM or sometimes in ROM) that they share with each other and with global application data, system data, application and kernel stacks, memory mapped I/O ports, and the RTOS kernel itself.

Classic RTOS Model: Maximum Exposure

This time-worn and familiar architecture, while simple, is highly exposed to corruption: runaway tasks can write over application code and data, accidentally write into peripheral device registers, and can corrupt kernel data structures and overwrite the kernel code. Tightly packed task stacks can easily underflow and overwrite one another, or charge downward through memory to corrupt the top of the heap or other data or code laid out nearby.

Granularity of Failure in Space and Time with an RTOS

At a higher level, this informally organized and highly exposed architecture presents two key challenges to code quality: scope of the failure itself, and association of second order failures with the primary event.

When an individual task or other software component fails, the scope of its failure is almost impossible to determine, let alone that it failed at all. Even when a failure is detected and recovery attempted the granularity of failure ends up being the entire system: Monitor code cannot usually safely restart tasks and the RTOS cannot recover resources dynamically allocated by failed tasks. The result is that recovery is most often accomplished through the brute-force use of watchdog timers that reboot the entire system or software induced panics.

Most often when a program goes awry, it does so silently, so an errant task can corrupt data and code anywhere in the RTOS system. With luck, the impact of such corruptions arises immediately (illegal instructions generating exceptions), but it is more likely that the damage will only surface at a later date – seconds, hours, or months later. When aberrant symptoms do appear, it will be extremely difficult to associate unexpected program behavior, whether subtle or crash-and-burn, with its original cause.

Built-in Reliability from the Linux Programming Model

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

While they share this virtual address space with the Linux kernel, they cannot overwrite kernel code or data. Since applications/processes cannot see one another (they reside in unique virtual address spaces), they cannot corrupt each other's data or code.

Granularity of Failure and Rapid Recovery with Embedded Linux

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

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

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

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

FAQ - Misconceptions about Embedded Linux

The goal of this document is to help developers consider for migration from Legacy RTOSs to embedded Linux, mostly by introducing challenges and benefits of such a move. 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 Legacy RTOS (like 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.

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 like Legacy RTOS, 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- 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. However, applications in telecommunications infrastructure and in mobile telephony can benefit from capabilities specific to Carrier Grade Linux 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

Conclusions

Migrating from proprietary/closed source/legacy RTOSs to Linux provides benefit of a modern and well supported platform that is proven for versatility of use cases supported and is highly reliable, secure and feature rich.

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

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.

Appendices

- William Weinberg, Moving Legacy Applications to Linux: RTOS Migration Revisited, 2014
- Iisko Lappalainen, Migrating from ITRON to Linux Concept White Paper, 2013
- MontaVista White Paper – Streamlining the Embedded Linux Development Process
- William Weinberg, Real-Time Technology for Embedded Linux The MontaVista Advantage, 2007
- 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.

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

Doc Id: MVWP-LG2LX1-061716