



WHITE PAPER

Migrating Legacy RTOS Device Drivers to Embedded Linux

January, 2009

Prepared by

**Bill Weinberg, Linux Pundit
for MontaVista Software**

TABLE OF CONTENTS

ABSTRACT	3
INTRODUCTION	4
WHO SHOULD READ THIS WHITE PAPER?	4
MIGRATION DECISION PATH.....	5
LEGACY LANDSCAPE.....	6
LEGACY RTOS – YOUR STARTING POINT	6
BARE METAL RTOS ARCHITECTURE.....	7
RTOS DRIVER APIS AND CONSTRUCTS.....	8
LEGACY I/O SEQUENCE.....	8
MIGRATING DRIVERS AND I/O CONSTRUCTS TO LINUX	9
LINUX DRIVER ARCHITECTURE.....	10
INTERRUPT PROCESSING AND DRIVER SEQUENCE.....	10
STATIC AND DYNAMIC DRIVER LOADING/BINDING	11
DRIVER SUPPORT APIS.....	11
DRIVER TYPES AND EXAMPLES	12
CHALLENGES AND CHOICES	13
THE MUTABLE LINUX KERNEL API SET	13
MODULES VS. KERNEL DRIVERS	14
KERNEL AND DRIVER DEBUGGING.....	15
MIGRATION RESOURCES	16
OPEN SOURCE PROJECTS.....	16
LINUX DRIVER PROJECT	17
PROFESSIONAL SERVICES ORGANIZATIONS.....	17
CONCLUSION.....	17

Abstract

With the accelerating pace of application development and ever more compressed product life cycles, the ability to reuse and migrate legacy software to new platforms is paramount. Migrating from legacy RTOS-based designs to embedded Linux presents a range of engineering challenges and also opportunities for refining I/O architecture and formalizing system interfaces.

This White Paper examines the particulars of legacy RTOS device interfaces and provides heuristics, resources and concrete examples of migrating this critical code to modern Linux-based embedded platforms. The White Paper reviews the architecture and conventions of drivers built for VxWorks and other legacy RTOSes, and gives developers straightforward guidelines for mapping code and constructs onto Linux 2.6. It examines options for kernel and user space implementations, highlights licensing implications, and presents migration and debugging tools and techniques.

Migrating Legacy RTOS Device Drivers to Embedded Linux

Introduction

Since the beginning of this new century, Linux has successfully unseated the leading legacy platforms as the new embedded OS of choice for a wide array 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.

Notwithstanding its position as the leading embedded platform, Linux must accommodate code coming from decades of legacy designs. Many projects remain locked into the legacy platforms because of

- Misconceptions about Linux architecture, capabilities and performance
- Budget constraints for migration engineering
- Concerns about carrying forward device support for both special purpose and COTS legacy peripherals

The purpose of this white paper is to address these and other concerns, and to help OEMs make a clear case for migration.

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.

Who Should Read This White Paper?

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

- Developers and Development Team Managers
- CTOs, VPs and Directors of Engineering
- OEM Product Management Teams

Migration Decision Path

The path from legacy drivers to Linux need not be a tortuous one. There are, however, important considerations that should guide you and your team along the way.

Figure 1. schematizes the decision path from legacy device code to up-to-date Linux device drivers.

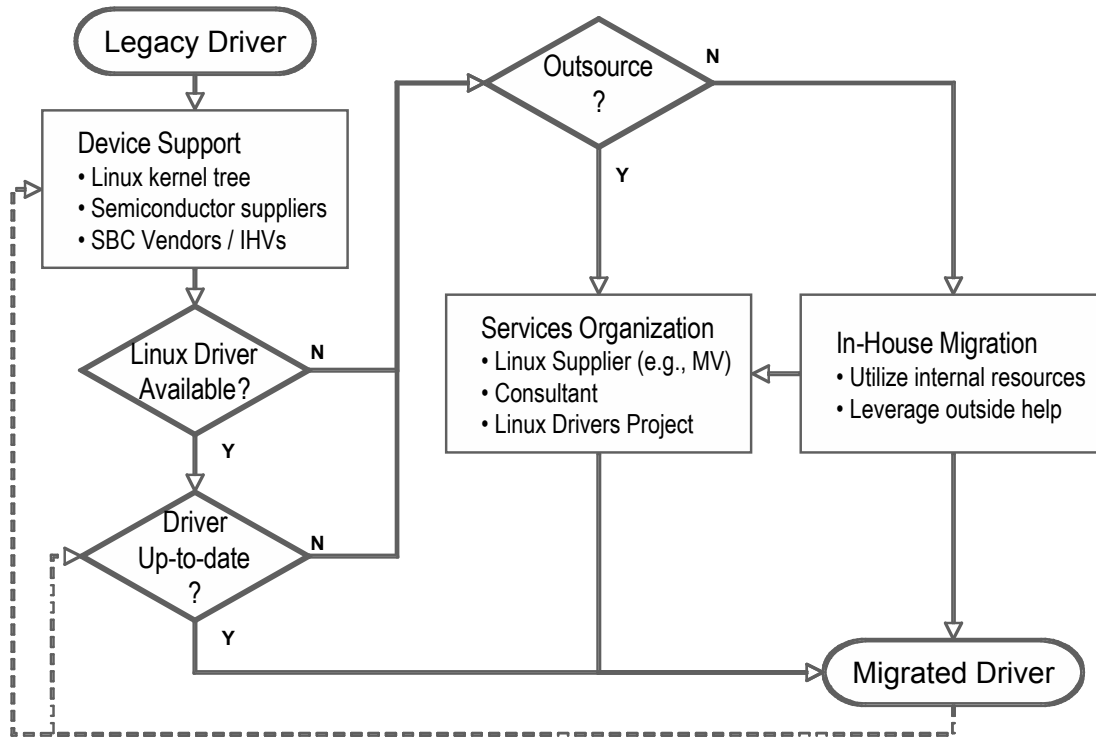


Figure 1. Migration Decision Path

Existing Device Support?

Even if you have the driver sources in your possession, it is worthwhile investigating to see if support exists already for the legacy device

- In the Linux Kernel Source Tree
- From the supplier of the device itself
- From Single-Board Computer (SBC) vendors deploying the same device

If a driver is available from any of these sources, you will still need to ensure that it is up to date with current kernel revisions and that you can actually build it.

Build vs. Buy

If no support exists from these communities and companies, you will need to consider whether your organization has the wherewithal to migrate the legacy code in-house, or whether you will need to contract with third parties.

Third party options include

- Linux Platform Suppliers (e.g., MontaVista Software)
- Individual Consultants
- Independent Professional Services Organizations
- The Linux Device Drivers Project (more later on this option)

With professional services organizations and with individual consultants, it is imperative that the supplier have genuine expertise (or access to it) in three areas: the legacy OS and driver model, the Linux kernel and driver architecture, and the particulars of the device itself as it fits into your application.

In-House Migration

The rest of this White Paper will focus on this path. However, even if you choose to apply internal resources to support the migration process, you can still leverage outside / third parties during the migration and, when it is complete, for continuation engineering and support.

Legacy Landscape

Before we address the key issue of migration, it is important to understand our starting point – legacy embedded OSes, their attributes and shortcomings.

Legacy RTOS – Your Starting Point

The complete roster of legacy RTOSes is long, dating back to two or three decades. Circa 1993, the author of this White Paper counted no less than 325 named embedded OSes from commercial entities, academia and free software project. Today, there are still several dozen in circulation, with a handful enjoying new product design wins.

- | | | | | |
|---------------|-----------------------|-------------|------------------|--------------|
| ▪ AMX | ▪ Integrity | ▪ Nucleus | ▪ REAL/IX RTexec | ▪ SuperTask! |
| ▪ AvSys | ▪ iRMX | ▪ OSE | ▪ REX | ▪ TargetOS |
| ▪ C Executive | ▪ iTRON / μ iTRON | ▪ OS-9 | ▪ ROM-DOS | ▪ THEOS |
| ▪ Chorus | ▪ LynxOS | ▪ PDOS | ▪ RT-11 | ▪ Threadx |
| ▪ CMX | ▪ μ cos | ▪ pSOS | ▪ RTXC | ▪ VRTX |
| ▪ eCOS | ▪ MQX | ▪ QNX | ▪ Skympx | ▪ VxWorks |
| ▪ Embedix | ▪ MTOS | ▪ QuickTask | ▪ SMX | |

Figure 2. – Legacy RTOSes

Legacy RTOSes can be segregated into two broad groups:

- Bare-metal executives, kernels and RTOSes bred for speed (VxWorks, pSOS, RTXC)
- MMU-enabled platforms, often UNIX-like, designed for robust interoperation (e.g., LynxOS, QNX)

We are going to focus on the Bare Metal group, in particular on the very widely deployed 5.2 version of VxWorks. It is, however, instructive to compare attributes of these OSes with Linux, especially as they apply to device interfacing:

	Bare Metal RTOS	Memory-Managed RTOS	Embedded Linux
MMU-enabled	No	Yes	Yes
POSIX Process Model	No	Sometimes	Yes
User Code Addressing	Physical	Virtual	Virtual
Kernel and Device Addressing	Physical	Virtual or Physical	Virtual
Driver Model	None/Partial	Formal	Formal
Interrupt Service	Anywhere	Driver	Driver Only
Driver Binding	Static	Usually Static	Static and Dynamic

Figure 3. – Comparing Legacy RTOSes and Linux Attributes

While memory-managed Legacy RTOSes share some attributes with Linux, and while many such RTOSes borrow driver source code *from* Linux¹, differences that impact migration can include

- Completely unique kernel driver APIs
- Non-paged, non-POSIX process models
- Unique driver address spaces, state models, and prioritization schemes (e.g., kernel threads)

The remainder of this White Paper will focus on Bare Metal RTOSes only.

Bare Metal RTOS Architecture

Most legacy RTOSes started out small. Design goals focused on providing essential synchronization and scheduling services, in a modest memory footprint, without getting in the way of applications and I/O. As such, classic Bare Metal RTOSes presented developers with a small API or system call set, and simple tools to implement Interrupt Service Routines (ISRs) or rudimentary drivers.

Over time, based on input from customer and from marketing, these simple executives grew in complexity. RTOS suppliers began adding APIs, new services and facilities; customers and third parties grew their software stacks to meet emerging application needs. By the mid 1990s, once Spartan Bare Metal RTOSes and the software stacks running on them began to resemble inverted pyramids: the size and complexity of software running over the RTOS greatly exceeded the platform’s capacity to manage it. As in real life, inverted software pyramids tend to topple, suffering from

- Simple kernels that cannot cope with complex software stacks
- Unstable or non-existent off-the-shelf middleware and application software

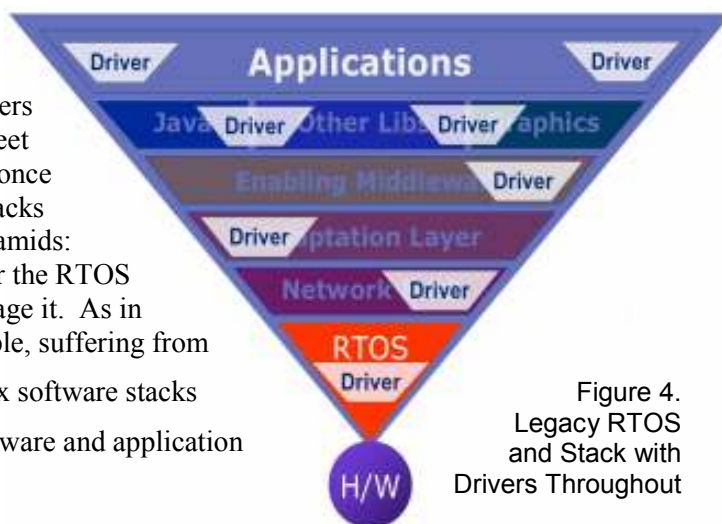


Figure 4. Legacy RTOS and Stack with Drivers Throughout

¹ Suppliers of these RTOSes have publicly stated that they use Linux device drivers as the starting point for their own device support.

- Inflexibility of a single-vendor solution
- No support for deployed memory protection
- Mismatch between modern CPUs and decades-old RTOSes
- Inability to support large teams of engineers and millions of lines of code

The aggregation of functionality and code over time and the lack of a formal device driver models led to the insertion and dispersion of device interface code not just at the kernel level, but throughout the legacy stack. Driver code could come from RTOS vendors, from chipset suppliers, and purveyors of networking stacks, file systems, Java virtual machines, graphics subsystems, other third parties and of course from in-house OEM efforts.

Co-mingling driver code with middleware and applications was probably not planned, but as part of the legacy landscape, it complicates re-use, debugging, maintenance, and ultimately, migration. Mixing system and user code ties it, sometimes inextricably, to its location, and can confound attempts at determining system vs. user context² during debug. Spaghetti code mixture of applications and drivers also complicates coding and building, requiring broad reentrancy (since drivers can also call user programs and libraries) and compilers must be directed to generate pre-ambls and codas for ISR vs. subroutines.

RTOS Driver APIs and Constructs

Legacy RTOSes boast dozens, sometimes hundreds of APIs. A relatively small subset of these APIs are relevant to I/O, but typically ALL APIs are available to legacy device driver code. In preparing your legacy driver code for migration, it is advisable that OEMs audit API use in driver code. If calls do not truly build on kernel-level functionality and have direct analogs in Linux user libraries, code using those APIs probably should be marked for migration into Linux user space and doesn't belong in a driver anyway.

Some legacy RTOSes do offer a “semi-formal” driver architecture. VxWorks 5.x, for its part, presents system and application developers with UNIX-like driver entry points that plug into VxWorks BSPs: `creat()`, `remove()`, `open()`, `close()`, `read()`, `write()`, and `ioctl()`. These APIs and the code that implement them map easily onto Linux driver architecture, but do not alone foster migration – underlying implementation and infrastructure are not necessarily similar: VxWorks and Linux each have their own kernel services API sets.

In practice, legacy RTOS driver models served RTOS vendors and their direct partners, but were usually not adopted by OEMs and other parties, who preferred *prior legacy* models and *ad hoc* approaches to I/O interfaces. The result is a legacy landscape with a majority of ad hoc device drivers surviving as source code, while vendor-supplied interfaces using formal driver models come down to OEMs in binary form only.

Legacy I/O Sequence

Figure 5. illustrates the ad hoc mixture of system and user code in legacy drivers. In this producer-consumer model,

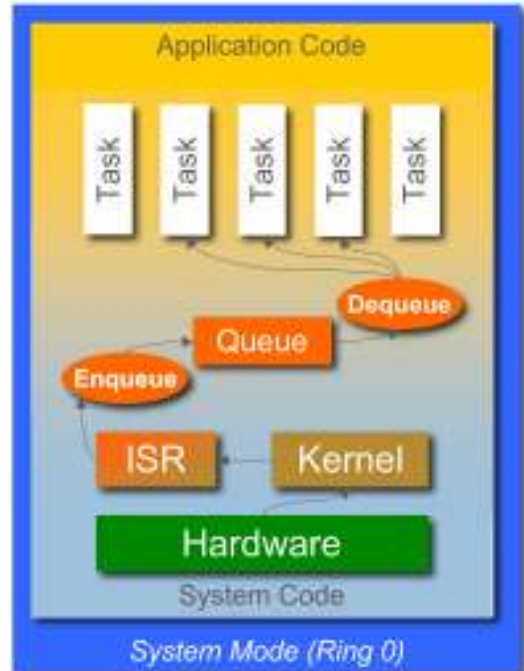


Figure 5. –Producer-Consumer Legacy I/O

² VxWorks actually has a macro for determining whether code is executing as an ISR or a regular program

an interrupt triggers entry into an ISR. As with formal driver “top halves”, the ISR performs basic processing and either completes the input operation locally or lets the RTOS schedule deferred handling. In some cases, deferred processing³ is handled by a user thread, usually an ordinary RTOS task. When and where the data ultimately is acquired (ISR or deferred context), ready data is placed into a message queue, for consumption by one or more application tasks.

Output occurs using comparable mechanisms—instead of using `write()` or comparable system calls directly, one or more RTOS application tasks places data into a queue. That queue then is drained by an I/O routine or ISR that responds to device-level ready-to-send interrupt, a system timer or another application task, performing I/O directly on data drawn from the queue.

Underlying this type of I/O model is the very straightforward interrupt processing sequence common to most bare metal RTOSes:

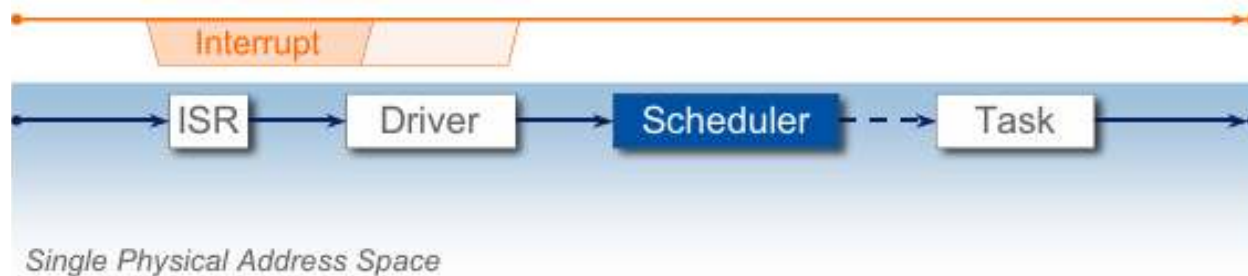


Figure 6. – Legacy RTOS Interrupt Processing Sequence

The figure above illustrates how upon receipt of an interrupt, the RTOS dispatches an ISR (given that interrupts are not masked by another ISR or the RTOS kernel). Optimally, ISR processing is short and sweet: acknowledging the interrupt and passing control to the “top half” of a driver that performs minimal processing and schedules the remainder of the I/O operation in a deferred context (a user or “kernel” thread). It is a common kluge, however, to preempt other device servicing by performing “real work” inside the top half itself, or worse, within the ISR.

Migrating Drivers and I/O Constructs to Linux

The good news for developers faced with migration from legacy to Linux is that core driver concepts from legacy drivers also exist for Linux. The great news is that while Linux I/O can appear very different from legacy drivers, the differences generally result in more structured and maintainable device interfaces. These differences include:

- Formal driver architecture – possible but not optimal (or recommended) to create ad hoc I/O code
- Richer interrupt processing sequence – more options for deferred processing
- Options for building/binding drivers statically to the kernel or loading them post-boot
- Driver APIs and name space restricted to the Linux kernel – no access to user code libraries (although common calls have available kernel versions)
- Infrastructure optimized for main driver types – character, block, network, graphics, etc.

Let’s briefly explore each of the above areas and how they relate to comparable functionality in legacy systems.

³ Deferring the bulk of interrupt processing and actual I/O operations to a non-interrupt context improves overall interrupt latency by rapidly unmasking interrupts and allowing I/O operations to take place with priorities harmonized with the rest of the system.

Linux Driver Architecture

Linux character and block drivers (typically for serial I/O, disks, etc.) present calling user code with five entry points: `open()`, `close()`, `read()`, `write()`, and `ioctl()`, which must be fleshed out in actual driver code. Following a file metaphor (as with desktop systems and servers), these APIs open and close an I/O stream, perform character or block reads and write, and allow special manipulation of the interface (via `ioctl()`).

The legacy producer-consumer model from the previous section would function as follows under Linux (see Figure 7.):

- User processes or threads perform a `read()` (can be synchronous/blocking or asynchronous/non-blocking)
- `read()` system call causes transition from user space to kernel context
- When a subsequent interrupt occurs followed by I/O processing, the driver writes resulting data into buffer space belonging to user process/thread context, available when the program resumes execution
- Rather than needing message queues to manage data flow, the Linux OS and libraries provide for buffering (which can be disabled if desired)

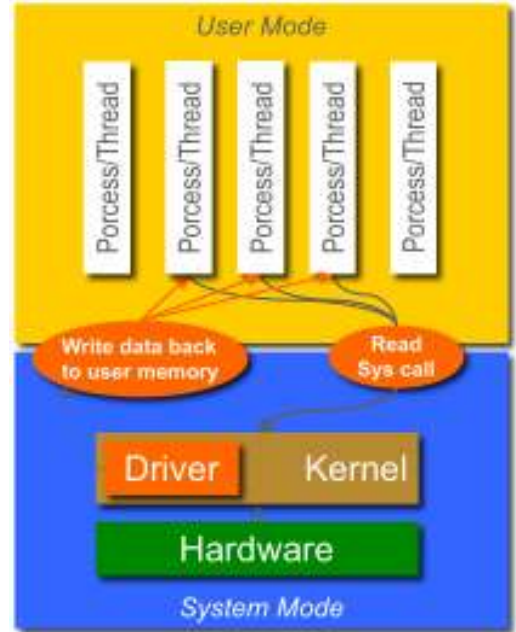


Figure 7. – Producer-Consumer Equivalent for Linux

Interrupt Processing and Driver Sequence

As with RTOS I/O processing, activity proceeds behind the scenes: an interrupt causes control to pass from ongoing execution into an ISR (except that the starting context may be in user space for normal programs, or in kernel space when other drivers or kernel code is executing). The ISR passes control to a driver top half, which performs lightweight processing and can then defer scheduling to any of several schedulable mechanisms:

Traditional Bottom Half

Before the 2.5 (experimental) kernel, heavy weight I/O processing always occurred in the driver “bottom half” or work queue, a deferred execution kernel context that runs immediately prior to scheduler execution. Bottom halves run with interrupts enabled and can be preempted by a new instance of the same top half that scheduled them. Older Linux systems exhibited non-deterministic behavior in part because kernel code running between the top and bottom halves was non-preemptible and because bottom half dispatch was FIFO ordered. Work queues are still the proper location for nominal priority I/O in 2.6 and beyond.

SoftIRQ

On return from *hard* interrupt processing (i.e., a top half), the Linux kernel checks to see if any *soft* IRQs have been raised. Note that only a fixed number of SoftIRQs exist and that they run in priority order.

Tasklet

Tasklets are kernel-resident functions that run in “software interrupt” context and are available to top halves for deferred execution, guaranteed to execute on the same CPU in SMP systems (to avoid accidental concurrency).

User-Space Process/Thread As with RTOS I/O, data crunching can always be deferred to user thread context after minimal driver activity. User space I/O processing should be treated as a slower bottom half activity; processing does not occur in interrupt context and is scheduled on a par with other user space threads. In theory, user-space interrupt handling is possible, but not optimal due to potentially long latencies⁴.

The interrupt handling sequence and the above mechanisms are illustrated in Figure 8. below.

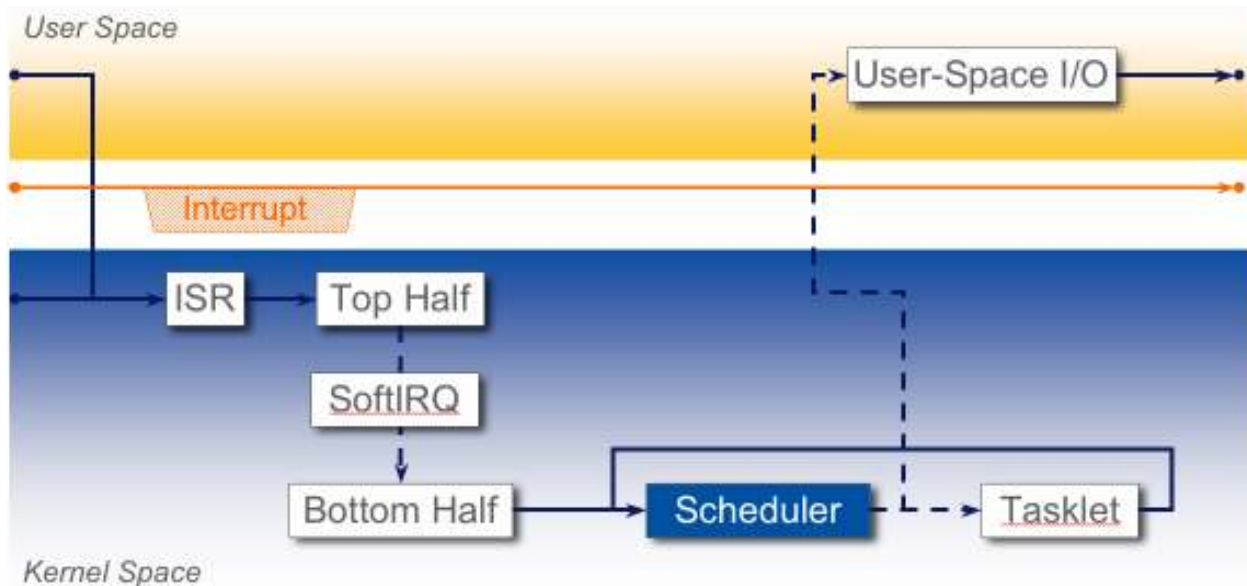


Figure 8. – Linux Interrupt Handling and Deferred Execution Mechanisms

Static and Dynamic Driver Loading/Binding

As with most legacy RTOSes, Linux driver code can be compiled into a kernel build, linked directly to the rest of the kernel name space and reside in the resultant binary image (`vmlinuz`). Unlike most legacy RTOSes, Linux provides the means to allow dynamic, post-boot insertion (and removal) of device driver code. The supporting mechanism is the module, a container for extensions to the Linux kernel.

Encapsulating a device driver inside a module results in smaller kernel sizes and potential faster boot times (smaller image to copy, fewer drivers to initialize), and can make its availability conditioned by application-level policy. However, modules also require a modicum of life-cycle programming for use – they must be inserted and initialized and in theory also shut down for removal. More about modules in the next section *Challenges and Choices*.

Driver Support APIs

For drivers linked statically to the Linux kernel, the entire namespace and symbol table is theoretically available to driver code. In practice, only a subset of those symbols and APIs are of interest to a given driver. For drivers encapsulated within modules, only kernel symbols and entry points exported with the `EXPORT_SYMBOL` macro are accessible. Note that modules, while separable from the kernel, must be built for a particular kernel version to ensure symbol table match and other types of compatibility.

⁴ See MontaVista Software white papers for hard data on Linux interrupt and preemption latencies.

You can view the exported symbols in `/proc/kallsyms`⁵. Note that when debugging is enabled, ALL kernel symbols will also appear in that `/proc` interface (for a 2.6 kernel, `kallsyms` can contain 40,000 entries).

Which APIs to use depends on the type of driver under migration. Resource allocation is accomplished with kernel-level analogs to calls like `malloc()` and `free()` – in the kernel, `kmalloc()` and `kfree()`. You can find documentation of these APIs for the 2.6 kernel in reference books like *Linux Device Drivers*⁶, but in Open Source Linux, the kernel source code itself is considered the primary and definitive manual (more in *Challenges and Choices* below). Most budding driver writers and migrants start with existing Linux drivers and modify them to meet the particulars of their application⁷ or apply knowledge gleaned there to migrate their legacy code.

Driver Types and Examples

From the need to support standard desktop and server hardware, Linux has evolved a range of driver types that can go beyond character and block drivers in scope and purpose. These types include bus drivers (e.g., PCI), graphics, frame buffer drivers, USB drivers, and network drivers. The last category is of significant interest, given the vast array of legacy NICs and MACs, and the need to support the rapidly evolving landscape of wireless interface types and protocols.

Networking Example

Many legacy RTOSes actually predate ubiquitous networking, and most don't even have a native TCP/IP stack – networking was “bolted on” after the fact, grafted into the RTOS-based stack by third party ISVs. VxWorks was an early adopter of TCP/IP networking, and integrated an early and highly customized version of the same Berkeley “BSD Lite” stack that exists in Linux.

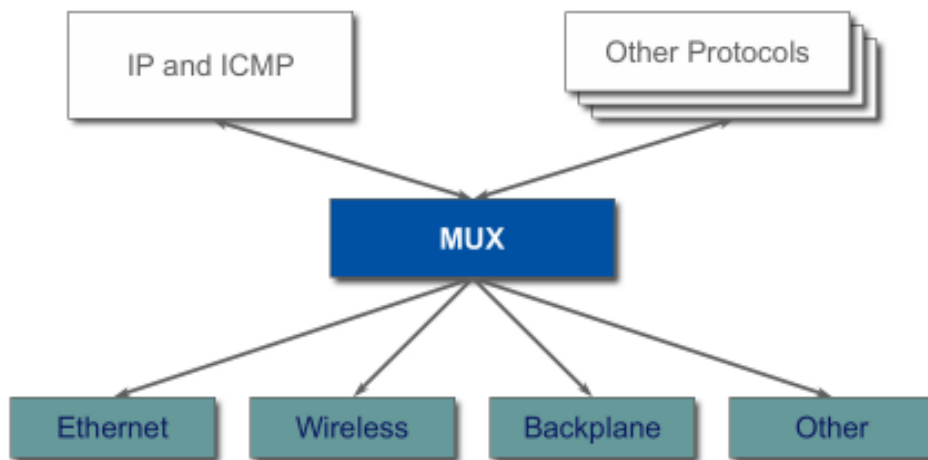


Figure 9. – Wind River VxWorks 5.x MUX Architecture

Some OEMs have found the networking stacks for VxWorks and other RTOSes less than suitable, and routinely replaced them with in-house or third-party equivalents. This should not be the case with the

⁵ For 2.4 kernels look in `/proc/ksyms`

⁶ Corbet, Rubini and Kroah-Hartman [2005]. O'Reilly Media. ISBN-13 978-0-596-00590-0.

⁷ Be aware that the resultant driver will likely constitute a derived work of the starting point. Reading over existing drivers to gain familiarity should not taint your company legacy IP.

Linux stack. First, its performance has been tuned by embedded, desktop and enterprise users for excellent throughput from intelligent marshaling of both large and small packets. Second, it is tied both architecturally and license-wise⁸ to the kernel, raising technical and legal barriers to replacing it.

Bringing legacy network interface drivers forward to Linux is possible but non-trivial. The VxWorks networking MUX (Figure 9.) provides a good point of departure. The MUX has well documented APIs for handing packets up and down in the stack and allows for plugging in non-IP protocols. The drivers underneath the MUX must implement specified callbacks, but are otherwise particular to the supplier – OEM, hardware supplier, ISV, or Wind River itself. As such, the principle of an abstraction from the stack above to physical layers translates perfectly to Linux, but the implementations do not. The Devil is always in the details.

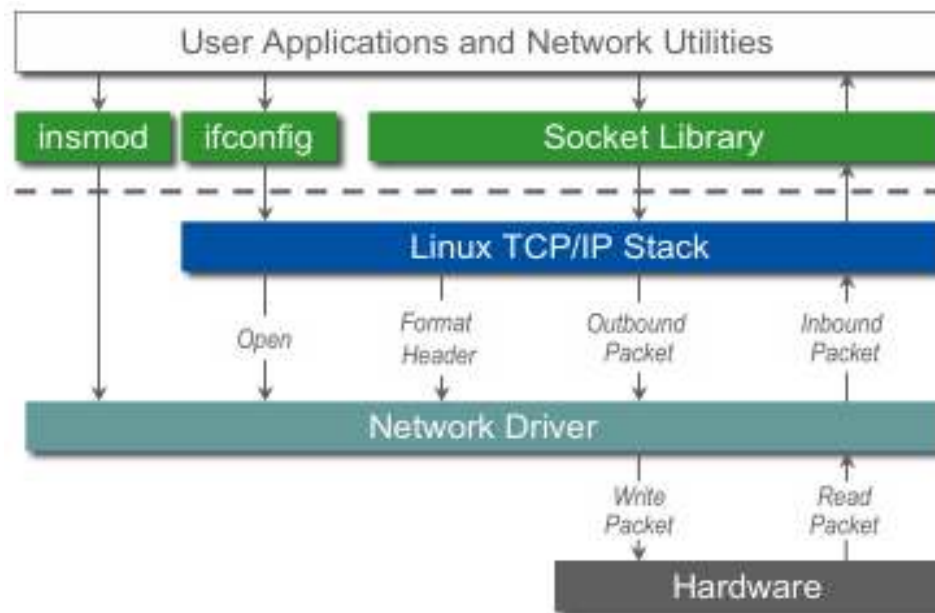


Figure 10. – Linux Networking Architecture (simplified)

Challenges and Choices

If migrating legacy code to Linux were trivial, you’d probably do it for kicks – you certainly wouldn’t be reading this White Paper! The migration process features a range of technical and logistical challenges, along with tough choices among alternative paths. Following are a few of these *Challenges and Choices*.

The Mutable Linux Kernel API Set

From the earliest versions of the Linux kernel onward, Linux creator Linus Torvalds pledged that users and user programs could rely on the stability of regular APIs. This pledge was easy for the Linux kernel community to keep, even as they tweaked system calls, since most of the user space APIs actually live inside library wrappers.

From those same early days forward, Linus also emphasized a commitment to kernel developers that they would be free to innovate and advance the kernel. In the almost two decades of Linux development and

⁸ A replacement TCP/IP stack would need to be licensed under the GNU General Public License version 2.0 to match the Linux kernel, or under a “compatible” license, e.g., BSD (as was the current Berkeley stack).

deployment, the kernel developer community has grown accustomed to getting its way when it comes to rapidly evolving kernel internal data structures and APIs.

Innovation vs. Stability

What this freedom to innovate means for OEMs is that Kernel/Driver APIs can change, sometimes radically, across kernel releases. This mutability is one factor that forces device drivers/modules to be built specifically to a given kernel version. While no great issue for source-based kernel drivers, the dynamic nature of the kernel API set makes building and distributing modules in binary form rather challenging: modules must be fully matched to kernel build, even if minor changes don't actually impact APIs in use, and require OEMs and their partners to align kernel and module versions with forward and backward porting.

This inability to “future proof” device drivers can create tension between OEMs and kernel developers. OEMs want and need to install new drivers on legacy products, are hesitant to upgrade the kernel for fear of breaking existing drivers, and don't look “upstream” for hardware support since the API status quo forces them to deploy and support stable if outdated versions of the kernel. Kernel developers want all deployers to use the latest (and presumably greatest) versions of the Linux kernel, and typically address bugs for future kernel revisions (they don't like to back port). They view OEMs reluctance to commit to public source trees as “hoarding” and not friendly to Open Source.

Best Practices

Rather than attempting to bridge the cultural divide, OEMs can follow straightforward best practices to mitigate risks arising from a dynamic kernel API set:

- Only use public symbols (EXPORT_SYMBOL) in both modules AND kernel drivers
- Work with the community / ecosystem to mainline driver code in order to ease forward migration, by bringing driver code into kernel.org or other stable trees (e.g., platform supplier or semiconductor vendor repositories)
 - When building on commercial Linux distributions (e.g., MontaVista Linux), standardize on the supplier's latest kernel version that meshes with your product life-cycle, limiting risk in your development process
- Work with the Linux Device Driver Project to make legacy device support mainstream (if there is a reasonable audience for the technology)
- Build in user-space as much as possible – it's more friendly to proprietary IP anyway
 - Leverage user space I/O mechanism like usb-lib
 - Run deferred I/O processing as user-space programs

Modules vs. Kernel Drivers

Choosing to build and deploy device drivers as kernel code vs. modules involves a mix of technical and legal/licensing considerations:

Modules and Licensing

Drivers built into and with the kernel are, as a matter of course, licensed under GPL version 2, just like the rest of the kernel. Modules, however, have a more ambiguous licensing disposition. While modules start out in user space (as files), they are not protected by the system call exemption in the COPYING (Linux Kernel License).

Community voices have been adamant that “Binary modules are illegal⁹”, but there is no reason to regard driver IP encapsulated in a modules as being a derived work and subject to GPL. Linus himself is ambivalent, wanting to bring in new functionality from legacy sources, but also trying to placate strident kernel developers. He has also stated that “There is NOTHING in the kernel license that allows modules to be Non-GPL¹⁰”, but following that same logic, Linus would be the first to admit that there is nothing in the COPYING that *requires* modules to be GPL, either.

	Kernel Driver	Module
Boot Image Size Impact	Yes – Code in Kernel	No
Boot Time Impact	Yes – Code to Copy/Init	No
Available at Boot Time	Yes	No
Insert/Remove as Needed	No	Yes
Available APIs	All	Exported Only
Prototype / Debug / Test	Rebuild Kernel	Rebuild Module Only
Driver License	GPL	Your Choice
Distribution Form	Source	Binary and/or Source

Figure 11. – Comparing Attributes of Linux Kernel Drivers and Modules

Best Practices

- License drivers/modules as GPL whenever possible
- Ship binary modules only if
 - You have clear, pre-existing copyright
 - You can show independent derivation of module code

Kernel and Driver Debugging

The greatest gaps to bridge between debugging legacy RTOS device drivers and those in Linux are

- The presence of an MMU and virtual addressing
- Maturity and focus of debugging tools
- Diffuse nature of documentation, with great reliance on the source code itself

Impact of MMU-based Execution

Debugging legacy application and system code with JTAG and other hardware and software tools, while not always a straightforward affair, at least enjoyed the presumption that *an address was an address*. In Linux and other MMU-enabled operating systems, virtual addressing enfolds applications and usually the OS kernel and drivers in constructed and constrained *logical* address spaces, formulaically related to the

⁹ Greg Kroah-Harman [2005], at OLS. However, there is no statutory prohibition in the US or elsewhere against binary distribution of modules and so *illegal* is probably the wrong term, anyway.

¹⁰ Linux Kernel Mailing List [2002]. October 17.

underlying *physical* memory. The Linux virtual addressing scheme is based on allocating pages of arbitrarily non-contiguous physical memory to create sequential (if not contiguous) logical address spaces.

This non-linear association of physical and logical pages of memory results in address translation by adding an MMU-determined offset to a physical page address in order to find its logical equivalent, and the converse as well. For debugging, it means that software and hardware tools must understand the physical-virtual dichotomy and be able to translate symbols, breakpoints, etc. on the fly.

Exceptions to and special applications of address translation include

- **Boot Time** - Initial boot phase occurs with physical addresses until MMU is enabled, after which ALL addressing is virtual
- **Page swapping** – virtual address can enable virtual memory, with code pages being swapped to disk or other storage at will. Tools need to comprehend the implications of swapping or can rely on user APIs to lock down code and data in physical memory
- **Hardware-based Kernel Debug** – debugging the kernel is actually simpler than controlling and instrumenting user space execution: kernel code pages are always contiguous in physical memory because they were allocated together at boot time. An “anchor” for JTAG debuggers has existed since at least the 2.2 kernel to facilitate source-level debugging.
- **Shared Libraries** - present special challenges, because while Linux allocates a single instance of the library, the mapping of library code into different processes can occur at unique addresses.

Maturity and Focus of Debug Tools

Linux relies primarily upon GNU tools like GDB and other open source projects like KDB for kernel debugging. These tools boast at least two decades of deployment and are often the same tools that developers used with legacy RTOS code (in some case before Linux even existed).

However, Linux kernel debugging, while offering full support for source-level debug, does not support “kernel-awareness” in the way that RTOS debuggers (e.g., XRAY+ for pSOS+) often did, comprehending, parsing and displaying kernel internals, synchronization states, scheduling information, etc. Linux kernel debugging, by contrast, is like debugging a large and complex stand-alone program. Look for kernel awareness in separate tools like

- **ps** and **top** – process and thread display
- **LTTng** – Linux Trace Tool Kit (next generation) for tracing system execution
- Commercial tools like Viosoft Arriba and MontaVista DevRocket

Migration Resources

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

Open Source Projects

- **V2Linux.org** – project maintained and recently updated by MontaVista with tools, libraries and documentation for migration. Visit <http://www.mvista.com> to learn more.
- **V2Lin** – Update to original V2Linux/p2Linux projects (circa 2001) : <http://v2lin.sourceforge.net/>
- **NDISwrapper** – Supports encapsulation, reuse of Windows network drivers; primarily useful for x86 PCI/USB NICs. More at <http://sourceforge.net/projects/ndiswrapper/>

- **User-Space Device Drivers** – User Level Device Drivers for Linux. Find the Gelato project at <http://www.gelato.unsw.edu.au/IA64wiki/UserLevelDrivers>
- **USB User Space Drivers** - libUSB is now mainstream; visit <http://libusb.wiki.sourceforge.net/>

Linux Driver Project

In 2006, a project was formed to augment and stabilize the device driver inventory for Linux and to assist IT organizations, OEMs and others in creating free and open source device drivers for a broad range of hardware. The project is led by Greg Kroah-Hartman and is endowed by the Linux Foundation. Over 200 project participants focus on creating three types of output:

- **Mainline device drivers** – community-maintained and residing in the kernel.org tree
- **Out-of-tree drivers** – following kernel standards but not accepted into mainline trees
- **Drivers needed** – backlog/queue and requirements tracking

Learn more at <http://www.linuxdriverproject.org/>

Professional Services Organizations

Stemming from its broad and deep enterprise deployment, Linux expertise is more widely disseminated than that for most legacy RTOSes (even VxWorks). Many options exist for finding outside help in the driver migration process:

- Linux distribution and tool kit suppliers
- Integrators and service providers
- Individual consultants

Caveats

Ubiquitous Linux knowledge is not the same as readily available expertise on migrating legacy RTOS drivers to Linux. Driver migration is a form of generic consulting! A qualified professional services professional or organization, to add value to a driver migration exercise, should boast expertise and experience in

- The Linux kernel in general and your embedded Linux distribution in particular, both commercial and in-house/RYO
- The architecture of the legacy RTOS (VxWorks, pSOS etc.)
- The device type and technology in question (networking, FibreChannel, etc.)
- Actually writing device drivers!

Conclusion

This White Paper has provided OEMs and integrators with a range of heuristics and actionable information to help evaluate the prospect of migrating legacy device code to modern embedded Linux. The goal of this document has been to elaborate architectural principles, elucidate resources and call out pitfalls and challenges facing teams contemplating and undergoing the migration process.

While migrating device driver code is by no means a trivial exercise, it need not be all-consuming. Community and commercial partners can ease the forward transition of what is essentially software infrastructure, ultimately freeing your company resources for creation and implementation of differentiating features and functionality in your next-generation product design.