Catalysts for your embedded software development

Embedded Linux Best Practices

This paper identifies and discusses best practices of Embedded Linux software development.

2 December 2006

Embedded Linux Best Practices

Table of Contents

Introduction

Linux is now commonplace in embedded systems. A company that is able to successfully use it often enjoys significant advantages over its competitors who use a traditional RTOS. But too many companies get into a lot of trouble when they switch to Linux, usually for reasons that crop up again and again. What do companies do to successfully adopt Embedded Linux?

This paper identifies and discusses best practices of Embedded Linux development. It should be useful for companies considering whether they should embark on an Embedded Linux development project and also those who have already done so but are finding it difficult. This document should be of interest to managers, project leaders and software engineers. Topics such as comparing RTOS/Linux software development practices, choosing hardware/software vendors and the implications of GPL licencing are discussed.

Is Embedded Linux Suitable?

Contrary to the hype that often surrounds Embedded Linux, Linux isn't always the best OS to use in embedded systems. How do you evaluate whether Embedded Linux is suitable for a particular product development?

First, understand why Linux has become so popular in embedded systems. The number of embedded applications where Embedded Linux is not suitable is becoming smaller and smaller. Today, it is only hard real-time applications that don't readily lend themselves to Embedded Linux. Even cost sensitive and small footprint applications like in-car audio systems, mobile phones, set-top boxes, TVs and network devices can be implemented using Linux. But why use Linux and not, say, BSD or Solaris? What makes Linux different from other UNIX variants is that it runs on a wide variety of CPUs. The Linux kernel sources are well structured such that CPU-specific code is easy to find and is minimised. As a result, Linux is supported on CPUs such as PPC32/64, Motorola PowerQUICC, ARM, Sparc32/64, SH, MIPS32/64, Intel X86 and even the MMU-less Motorola 68K and ARM variants. Linux can be made to run on almost any CPU board. As CPUs get faster and memory gets cheaper and larger, software gets more complex. Today, few companies write all of the software that goes into their embedded products. When there's a pool of free UNIX software that would add features to a product, it is trivial to add it in when the OS being used is Linux. Porting such code to RTOS can be relatively time consuming so by building embedded systems using an OS that is already supported by most open source software, complex embedded products may be built with less development effort.

A common myth about Embedded Linux is that it is slower than RTOS solutions on the same hardware. The author of this paper has ported a number of small embedded boards to Linux (e.g. Motorola MPC860 CPU with 8M RAM and 4M flash) and in each case, Linux outperformed the existing VxWorks and Nucleus implementations by about 10%.[1] The memory footprint was about 20% larger than that of VxWorks once the system was minimised. However, if Linux is used as if it were an RTOS, with little effort spent on taking advantage of what Linux has to offer, the Linux solution will be considerably slower than the equivalent RTOS solution and have a memory footprint perhaps twice as large. It is important, therefore, to understand what is different about Embedded Linux when compared with traditional RTOS *before* starting an Embedded Linux development. Also keep in mind that Embedded Linux may not be the best choice for a particular product.

When deciding whether to use Linux, consider the following questions:-

	Question
1	For off-the-shelf hardware, does Linux already run on the board? For custom hardware, do device drivers already exist for Linux for the devices used?
2	Should a commercial Embedded Linux distribution be used? If not, who provides support?
3	How will local software development processes change if Embedded Linux is used?

[1] Both boards were small, home DSL routers. Performance was measured using packets per second forwarding rates of minimum sized packets, tested using commercial packet generators.

4	What software tools are available for Linux?
5	What are the implications of GPL? Will using Linux force proprietary code to be released under GPL?
6	What are the real-time constraints of the system?
7	Does each Linux CPU have a serial port and an ethernet port available? If not, can they be added as optional fit components for development boards used by software developers?

Asking these questions is the first step towards a successful Embedded Linux development. It is quite common that Embedded Linux seems to be the obvious choice in all but one or two areas. Focus the analysis on those areas.

Choosing a Hardware Vendor

Few hardware vendors do a really good job at providing quality Linux device drivers and kernel distributions for their embedded hardware. To maximise the benefits of Embedded Linux, it is important to choose hardware vendors, devices and boards carefully, especially when there is little in-house Linux expertise.

In some cases, hardware vendors hack changes into a particular Linux kernel version with little thought. For them, this is a tick-the-box exercise – "Yes, we support Linux". When a hardware vendor provides a complete Linux kernel, it is quite common for their new device drivers to be implemented in a way in which the drivers do not use standard core Linux subsystems such as PCI, USB or I2C. Worse, the vendor scatters hardware-specific code around common kernel code rather than hook it into the kernel's own hardware support subsystems. This is almost always a sign that the hardware vendor has taken an RTOS driver and has tried to port it with as little effort as possible to Linux. For the majority of devices, a Linux device driver should not need core kernel changes. When such changes are needed (for example, where the device is of a new class that has not yet been supported in Linux), one would expect to find discussions in Linux mailing lists about how best to implement a driver for the new device.

One of the reasons for switching to Embedded Linux is the wide range of device drivers and hardware support which is available out of the box. If the choice of device is flexible, best practice is to evaluate devices with drivers already in standard Linux kernel distributions and then use one of those devices in the hardware design. This inevitably requires software engineering effort early in the project, helping hardware engineers choose devices with best Linux support.[2]

The same is true when choosing an off-the-shelf CPU board, when alternatives should also be carefully evaluated with regard to Linux support. Some CPU boards are supported by the standard Linux kernel distributions. These will almost always indicate that the Linux support

[2] Time spent early in a hardware design to minimise software development effort is always beneficial. This is even more important when a company has little in-house Linux device driver expertise.

for such boards is good because the hardware vendor has spent the effort to submit kernel patches to the Linux community and worked to incorporate feedback from reviewers. When looking for a suitable CPU board, therefore, the first place to look for viable alternatives should be the Linux kernel source tree itself. Board support code is always under an architecture-specific directory, e.g. `arch/ppc/platforms`. For newer boards, it is also useful to check the Linux kernel mailing lists for evidence that the hardware vendor is working on having support for the board included in subsequent kernel releases.

Sometimes, device drivers for a particular device or board are available only from a commercial Embedded Linux vendor. These drivers have not been through the rigorous Linux kernel code review process. However, they are usually of reasonable quality because Embedded Linux specialist companies live and breathe Linux. When device and/or board support is available only from a commercial Embedded Linux vendor, it may be beneficial to choose that vendor to provide the Linux solution.

But things aren't often so simple. In cost sensitive markets such as high volume, consumer devices, device choice often comes down to unit cost. When a company operates in those markets, it is essential to have good in-house or outsourced Linux engineering skills in order to be able to develop Linux drivers for devices or boards that don't already have Linux support.

To summarise, when looking for a Linux-friendly hardware vendor, look for the following:-

- Look for device or board support of one or more boards from the hardware vendor in the standard Linux kernel source tree.

- Look for device or board support provided by the vendor as patches against a standard Linux kernel. When the vendor does not provide patches and instead only provides a complete kernel source tree, they are sometimes trying to hide the quality of their kernel changes or are encouraging the use of their specific kernel. Consider the implications of accepting a complete kernel source tree rather than patches to a standard kernel. In the case that you wish to update the kernel to a later version, perhaps for device or feature support available only in a later kernel version, either the vendor must port their changes forward to the new version, or you must take their changes forward. The vendor may be unwilling to do this work, and in this case you will be burdened with understanding and reapplying the vendor's changes to the new kernel. Some hardware vendors provide no assistance to their customers to help them understand what changes were made to the standard Linux code in order to make it work.

- If it is anticipated that some device driver or kernel board support development work will be required, check that the hardware vendor provides sufficient hardware documentation (device register bit definitions etc) to do the work. Some hardware vendors are not friendly towards open source development. The Linux kernel is licenced under GPL which means any changes made to it must be published, though it is possible to implement binary-only drivers as loadable kernel modules to avoid GPL in some cases. It will sometimes be necessary to develop binary-only drivers where the hardware vendor protects its devices under NDA. If the hardware vendor provides some OS-agnostic C routines for their devices, check the licence of such code since it may also prevent that code being used in an open source kernel. The subject of GPL licencing is discussed in more detail later.

- Look for drivers developed in the community. Search the Linux kernel mailing lists or architecture specific mailing lists. Don't assume that the driver is of good quality though – the standard of Linux device drivers varies enormously, particularly those that have not been integrated into the kernel proper. Examine the driver source code. If it wouldn't look out of place in official Linux kernel distributions, it is usually a sign that it is of reasonable quality as the author studied Linux code before writing it. Be wary of kernel code that looks like it can also be compiled for other operating systems. Such kernel code tends to come with its own implementation of various kernel subsystems (e.g. PCI bus address assignments, I2C, hardware device access mechanisms) in order to make it portable to other OS's.

Choosing a Linux Distribution

Having identified the best hardware, how should a Linux distribution be selected?

Sometimes the hardware vendor will partner with a commercial Embedded Linux vendor and will offer Linux support for only that distribution. Sometimes there may already be Embedded Linux development projects within the company, so using the same vendor for the new project may be the beneficial. In those cases, there may be no real choice. We will deal with the case where there is a free choice of distribution here.

The first decision is whether to use a commercial Embedded Linux vendor or to roll your own from scratch. Some are surprised to learn how straightforward it is to build a Linux system from resources available for free on the Internet. Pre-built cross-development GNU tools are now easy to find and the kernel may be cross-compiled out of the box. Applications can be harder to cross-compile, although there are now fully contained build environments such as buildroot (http://www.buildroot.org) which can build up an entire filesystem of applications from a single invocation of make. Other open source projects aim to build Embedded Linux distributions to suit specific requirements, e.g. http://leaf.soureforge.net. Without sufficient experience of Linux and GNU tools, a prudent choice would be to go with a commercial vendor. In the longer term, however, it can be very beneficial to be vendor neutral.

There are several commercial Embedded Linux vendors fighting for the same business. The following table lists the most popular ones. There are many other vendors – too many to list here.

Vendor	Website
Montavista Inc	http://www.mvista.com
Timesys Inc	http://www.timesys.com
Wind River Systems Inc	http://www.windriver.com/linux
Denx Software Engineering Gmbh	http://www.denx.de
Sysgo AG	http://www.sysgo.com
FSM Labs Inc	http://www.fsmlabs.com

Embedded Linux vendors are obviously keen to win new business. So talk to them, if only to make an informed decision about what you get for your money. Keep in mind though that using a commercial Embedded Linux distribution won't guarantee success. It can even be advantageous to roll your own since it provides optimum flexibility and avoids possible vendor lock-in. Best practice is to evaluate options, ideally with a trial project. The engineering effort expended in learning how a Linux distribution is put together will rarely be wasted and will probably pay dividends throughout the project.

The Linux Paradigm Shift

Linux changes the rules for embedded software development. It is important to accept this and change traditional RTOS development practices accordingly.

Software Development Methods

Companies used to developing with commercial RTOS solutions often have difficulty adjusting to Embedded Linux. Some of the advantages of Linux such as separate process address spaces and separate kernel are sometimes perceived as disadvantages, getting in the way of a designer that wants to program the RTOS way, where everything is implemented in one *big blob*. Linux encourages robust software design, identifying major functional blocks and how they would communicate with each other, thus leading to a number of user processes and possibly new kernel device drivers. When the tasks are decomposed in this manner, software can be compiled and tested in smaller pieces, thereby decreasing the edit / compile / debug cycle. These separate components also lend themselves to natural software reuse. This is just good software engineering. Best practice is to design the software as a typical Linux box, with the system implemented as multiple application processes using standard kernel device drivers and subsystems.

Just as in the RTOS world where developers might specialise in Board Support Packages (BSPs), device drivers or applications, Embedded Linux has the same skill separation. RTOS BSP developers are gurus in low level programming and board bring-up. Similar skills are needed to do initial board bring-up of Linux on an embedded board. RTOS device driver developers need to know about how to access hardware and use low level OS primitives to implement the driver. Again, Linux device driver developers know about internal kernel services and device-specific APIs in order to implement new drivers. Finally, RTOS application developers work at task level, using queues, mailboxes etc to communicate with other tasks. And surprise, surprise, Linux also has application developers. But this is where there are differences between Linux and the RTOS world.

In Linux, or any UNIX OS for that matter, applications interact with the kernel using well defined APIs and programming methods. This contrasts with the RTOS equivalent, where device drivers typically provide a library of calls which are specific to each driver. Some RTOS APIs might be more formal than a plain C API, such as Wind River's SENS network drivers, for example. However, the key thing is that in an RTOS, an application task can call any public C function if it wants to, including any device register access routines. In a Linux system, this is not the case. When using Linux, therefore, much more care is needed with the software system design to ensure that functionality is implemented in the appropriate place.

Another difference is that application developers have more debug tools at their disposal: the GDB debugger, memory leak and bounds check tools like dmalloc, valgrind and syslog trace messages are commonly used. Moreover, user applications may be easily restarted without having to reboot the target system, thus reducing the compile/debug cycle. Linux kernel developers must work in a very different way, sometimes without a debugger[3]. Any program failure can often lead to system lockup, requiring a reboot, which adds to the perception that kernel programming is difficult.

Some shy away from Linux kernel development because of its perceived difficulties. This is a mistake. Although there is a steep learning curve, an experienced kernel developer can implement functionality just as quickly, if not more quickly than someone implementing a similar thing in userspace. Obviously, only certain functions should be implemented in the kernel, mostly those functions where there needs to be close interaction with a hardware device or low-level kernel subsystem.

In general, RTOS BSP and device driver developers are well suited to working in the Linux kernel and RTOS application developers work best with userspace applications. Best practice is to organise project teams along these lines, encouraging developers to become board bring-up, kernel development or application development specialists.

RTOS to Linux

The RTOS model lets software developers write task-level applications which export external C APIs for other tasks to use. These are plugged together to make one big application. If the software architecture isn't quite right, public APIs between the components may be hacked to workaround design problems. With Linux, if the components are separate processes or interfaces cross the userspace-kernel boundary, changing such interfaces might be more difficult or even impossible. Linux not only encourages good software design, it requires it! When writing software targeted for UNIX/Linux, it must be designed such that the software takes account of the separate kernel and userspace process memory model. UNIX/Linux doesn't work under the hood like an RTOS, so when porting software originally designed to run on an RTOS, it must be carefully audited (and often redesigned) to run efficiently on UNIX/Linux.

Having invested a lot of effort in proprietary RTOS code, companies are often reluctant to discard their existing RTOS software modules in Linux products when Linux has equivalent functionality built in. Examples are PCI bus scanning, I2C, USB and networking protocols. It should be obvious that a subsystem designed for Linux will be a better fit than one designed for a different environment. It is best to use the Linux subsystems in favour of equivalent RTOS subsystems, no matter how painful this may seem.

When moving from RTOS to Embedded Linux, it is important to ensure that all developers understand that the way they develop for Linux is very different from RTOS development. Developers should be encouraged to subscribe to Linux mailing lists covering the areas in which they work and evaluate Linux implementations of features with which they are familiar. For example, an RTOS developer who has written several I2C drivers should study the Linux I2C kernel code and applications that use it. Developers moving from RTOS environments may have difficulty understanding why a different approach has been taken on Linux. Experimentation with a working system and reading the source will reveal a lot. Best

[3] Some kernel distributions include an in-kernel debugger such as KDB or KGDB. These are usually installed as kernel patches.

practice is to design for Linux -- don't try to hide developers from the differences between Linux and RTOS environments.

Software Portability

If software is targeted at Linux, does this make it difficult to port to other environments where Linux is not used?

Applications

As far as portability is concerned, Linux applications are readily ported to other POSIX environments when they are written to POSIX APIs. Some RTOS such as LynxOS and QNX are UNIX-like and provide true POSIX compliance. VxWorks also has POSIX wrappers. If new code were written to POSIX APIs, code could easily be ported to such RTOS's when required. However, since Linux runs on many different processor architectures, Linux code is inherently portable to many embedded platforms. It isn't unusual for companies to build a variety of products using different hardware architectures, all running a lot of common code under Linux.

POSIX has been designed by committees of many software experts and as a result, it is appropriate for many software environments. Its API covers most needs so can map to Linux, Windows and RTOS environments. Unfortunately, the reverse isn't true – APIs designed for RTOS environments seldom map to Linux.

Kernel and Device Drivers

The Linux kernel itself is very portable and configurable. In fact, porting Linux to new hardware can take less time than the equivalent task using an RTOS when the board's CPU type and serial port device is already supported by the core kernel. Once the kernel is up and running, device drivers can be tested. Many devices already have device drivers for Linux, even if they aren't distributed with the core kernel. The modular architecture of the kernel and its well designed internal mechanisms mean that device drivers are often usable on architectures other than those for which they were originally designed. The key point here is that investing effort in writing Linux kernel device drivers correctly yields device support for a number of different platforms when Linux is used on each of those platforms.

Some companies are understandably reluctant to write device drivers for a single operating system. Instead, they prefer to add a level of abstraction between the driver code and OS to enable a different OS to be used without altering the driver code. The same aim can be achieved by writing the core code of the device driver as a set of ANSI C, non-blocking functions which handle all of the low-level hardware accesses. These functions present a C API to the device. Then, an OS-specific driver is written which calls the common ANSI C routines. However, even this method still has its drawbacks.

- The Linux kernel provides primitives for hardware access, taking care of things like endian conversions, cache coherency, memory barriers, CPU pipelines, optimised data copying etc. These would be unavailable to the OS-agnostic driver functions.

- Developers learn a lot by studying device drivers in the core kernel and can quickly pick up techniques used. Implementing a policy of writing core code of device drivers as portable C functions tends to discourage developers from studying existing kernel code. Worse, the code might already exist in the kernel, but because the

developer isn't familiar with the Linux kernel source code, the existing code has been missed. Significant effort can therefore be wasted, potentially costing the company in time to market.

It is the author's view that device drivers should always be targeted and therefore optimised for the OS being used. It doesn't take long to port a driver to a different OS; the advantages of having some common code shared by drivers for different OS's are often cancelled out by the practicalities of software development: fixing a bug in a device driver for one OS may introduce a bug when the code is used in another OS. Device drivers are difficult to regression test and it is common for device driver developers to have specific knowledge of only a subset of the number of different OS's using a section of code.

Tools for Embedded Linux

People who are used to supported, proprietary tools provided by RTOS vendors are often disappointed by the tools available to support development for Embedded Linux. There are several reasons for this.

- Most Embedded Linux tools are Open Source and run on UNIX/Linux development hosts. Many RTOS developers are used to working on Windows development hosts and lack of familiarity with the environment makes the tools seem harder to use.

- Because of the wide range of applications that Linux is able to execute, Embedded Linux systems are much more complex under the hood than any RTOS environment. Tools are therefore much more difficult to develop. Separate memory address spaces for user processes and kernel and the way that the OS pages memory in and out make analysing the system externally very difficult for any development tool.

- RTOS developers sometimes expect to find exact equivalents of RTOS tools available for Embedded Linux. Tools like MemScope for analysing memory usage, for example, don't work for Embedded Linux because the virtual memory model breaks assumptions made by those RTOS tools. In Linux, a user application automatically cleans up when it exits, including any memory that it allocates. So many simple applications don't bother freeing allocated memory or closing files when the application is short-lived. When developing long-lived applications (e.g. daemons that run all the time that the system is up), developers must use good programming practices to find memory leaks, timing bugs, data bounds check errors etc by unit testing rather than relying on OS tools to find the problems during integration.

IDEs

Some commercial Embedded Linux vendors sell Windows tools supporting Embedded Linux, capitalising on those companies reluctant to throw away their Windows development environment. But these tools try hard to shield the developer from Linux characteristics, providing features such as the ability to configure and compile the kernel from a Windows GUI for example; in the long run, there is little or no advantage in these features. When developers gain familiarity with Linux, they tend to write better code for Embedded Linux systems and are usually able to debug problems more efficiently.

In the Embedded Linux market, the Eclipse IDE has become very popular. Commercial Embedded Linux vendors such as Timesys and Wind River build their own tools around the Eclipse framework. Some independent hardware/software tool vendors have even abandoned their own Windows GUIs in favour of providing Eclipse plugins. But IDEs are a personal choice; what suits one developer might not suit another.

Embedded Tools

Two of the most common, freely available tools used in Embedded Linux are Dmalloc and LTT.

Dmalloc is used to find memory leaks, bounds check errors and uninitialised data bugs in userspace applications. It provides configurable, runtime error checking and can be used to test cross-compiled applications running on the target hardware. It is good practice to test for memory leaks etc before integration testing.

LTT (Linux Trace Toolkit) provides a graphical event trace of what the system is doing during a time window. LTT provides similar capabilities as Wind River's WindView for VxWorks.

If software can be tested on development hosts in a simulation environment, other more powerful tools can be used, some of which are commercial. Such tools tend to be more powerful than those that run on the embedded target and they often have GUIs which make them easier to use for some. For this reason, best practice is that where possible, software is developed in such a way to allow it to be run in a simulated environment on a development host. This is made easier when the software design uses standard Linux subsystems such as network sockets for networking, framebuffers for video/graphics, V4L for multimedia/TV and so on. Off-the-shelf hardware may then be used in standard PCs to test most application code before it runs on the target hardware.

Many companies expect to do source level debugging of entire Linux systems using a JTAG probe with its proprietary debugger software. Because Linux systems are made up of several, independent applications and a kernel, debugging at system level requires more than a JTAG probe. In fact, with Linux, a JTAG probe can often only be used to debug Linux kernel code and even then, many RTOS JTAG debuggers are totally confused (and therefore useless) once Linux enables the CPU's MMU. Other tools such as memory usage and CPU usage analysers work very differently from their RTOS counterparts. Unlike RTOS tools which can be used to do basic debugging at a system level, Linux system-level tools are more appropriate for analysing system behaviour rather than basic debugging. With Linux, it is much more important to test individual components in isolation (including finding memory leaks) *before* bringing them together in the final system. This is just good software engineering.

Real Time Constraints

It is often said, especially by commercial RTOS vendors, that any system with real-time constraints is not suited to Embedded Linux.

The use of RTOS in embedded systems is largely historical. Only ten years ago, embedded hardware typically had a low-spec CPU with just enough memory to run the application firmware. As a result, the embedded firmware was often hand-crafted for each product. These systems ran an RTOS not because the system had real-time constraints, but because only

RTOS code would fit in the available RAM. As CPUs became faster and memory became bigger and cheaper, running a UNIX or Windows-like OS on embedded hardware became a real possibility. Linux was ported to embedded PowerPC CPUs as early as 1997 and now runs on almost every modern CPU, including MMU-less ARM and M68K. It is therefore important to understand whether the system being implemented has any *real* real-time constraints or whether an RTOS was used in similar products only for the reasons as outlined above.

Real-time constraints influence system software design, especially when Embedded Linux is used. But the term *real-time* is often misused when discussing OS behaviour. There are two specific cases to consider:

Real-time Type	Notes
Soft real-time	This is where specific time constraints exist, e.g. event X must be handled every 10ms, but failure to do so isn't catastrophic and the system is able to recover if the deadline is missed. Such failures shouldn't happen too often and the software should be designed to minimise that possibility. An audio player, for example, has soft real-time constraints on pulling data buffers for playback. If a buffer isn't ready when it is needed, an audio glitch occurs but the system continues to operate.
Hard real-time	This is where there are scheduling deadlines that the software absolutely must meet in order to maintain system operation. Failure to meet a hard deadline results in catastrophic system failure. A manufacture control system is an example of a system having hard real-time constraints.

Soft real-time requirements can be minimised by hardware design, e.g. interrupt queues, DMA request rings, FIFOs etc. Modern network hardware, for example, uses these techniques to minimise real-time constraints on the OS software. Event response latency is critical for adhering to soft real-time constraints. There are several low latency kernel patches from Montavista and RedHat to improve latency in the Linux kernel, along with other improvements in the 2.6 kernel. More recently, changes incorporated in the 2.6.18 kernel add interrupt pre-emption support and high resolution timers.

Where there are hard real-time constraints, however, more complex changes are needed to Linux. There are two Linux add-ons, RTAI and RTlinux, each aimed at making Linux fit hard real-time. They work by running the Linux kernel and all of its processes as threads under a custom real-time kernel. This changes the way Linux works and can sometimes negate one of the advantages of moving to Linux in the first place, namely being able to take advantage of the wealth of Linux drivers that are available. However, RTlinux from FSM Labs in particular is making significant inroads into Embedded Linux market share and might be a good choice where hard, real-time constraints really exist.

In Embedded Linux, best practice is to carefully evaluate all real-time constraints of the system early in a project development and to design the hardware and software to minimise those constraints. Software that services real-time events should be prototyped to measure system performance parameters such as interrupt latency and event handling response times.

Simple test applications are available to put the CPU under load, emulating operational environments before the product's application software is available.

Threads

POSIX.4 defines userspace APIs for semaphores, mutexes, queues, timers, extended signals, asynchronous IO and real-time threads. It is possible to map many of the common RTOS primitives to Linux using these APIs, resulting in an RTOS application running in a single process on a Linux kernel. This can be useful to get legacy RTOS code running quickly on a new OS but it seldom yields production quality systems. There is unfortunately a strong tendency for companies used to developing with RTOS to use POSIX.4 mechanisms and run everything in one Linux process. Such implementations are always slower and have a far bigger memory footprint than the equivalent RTOS implementations.

In UNIX, a thread runs in the same memory space as other threads of the application process. Threads therefore map RTOS tasks. However, threads should be, as their name implies, separate parallel threads of execution. In other words, the number of threads should be no more than the total number of parallel work items being processed. Yet threads are instead often used as a design modularisation technique, e.g. a thread to read device temperature every second, another to hit a watchdog periodically and another to wait for and process input events. A good UNIX design will minimise the number of threads. Until recently, Linux's thread support was relatively poor. The New POSIX Thread Library (NPTL) solves some issues like scalability but it isn't yet supported by all architectures. Also, NPTL is available only for 2.6 kernels.

There are other drawbacks with using threads.

- Many standard library calls are not thread-safe. It may be necessary, for example, to wrap up calls to malloc() and free() with mutexes to make them thread-safe.

- Debugging multi-threaded applications can be very difficult. GDB's support for threads varies from CPU architecture to architecture. A debugger can interfere too much with the scheduling of threads anyway – debuggers can sometimes cause more problems than they solve when debugging multi-threaded applications.

- Threads share the same memory space as all other threads of the application process, which means a bug in one thread can lead to obscure memory corruption problems seen in another (apparently unrelated) thread. When it is useful for more than one parallel thread of execution to share the same memory space, threads are useful. However, when threads are independent, that is they don't share data, they should either be implemented as a separate process or the application should be restructured to service queued work from a single application main loop.

POSIX.4 provides a real-time thread facility where a thread may be assigned a scheduling priority. A common bug when using real-time threads, however, is caused by a difference in behaviour of Linux real-time threads over RTOS tasks. In Linux, a real-time thread is given the CPU while it is the highest priority runnable real-time thread. The Linux scheduler trusts it

implicitly.[4] Since the standard Linux scheduler is not pre-emptive, a thread/process is descheduled only when it makes a blocking system call or when its time slice expires. Real-time threads have no time slice so if they loop without doing any blocking system calls, they lock out the entire system, including all other non real-time processes and kernel threads.

UNIX systems, including Linux, do not have many multi-threaded applications. Instead, complex applications are designed to spawn separate processes to handle parallel tasks, or are designed to avoid parallel tasks altogether. A lot of RTOS code uses multiple tasks (threads) unnecessarily. Unfortunately, this style of programming can often continue after a company chooses to switch to Embedded Linux.

It is a mistake to treat POSIX.4 like an RTOS API. Best practice is to use threads only when the system design and software architecture requires it.

Obtaining Support

Almost all companies who use commercial RTOS solutions expect to pay for support. Project managers are sometimes uncomfortable not paying someone for support so go looking for a company to provide Embedded Linux support services. Yet in many cases, problems can be solved by searching the Internet or asking questions in Linux mailing lists, especially if the developer works with the community on solving the problem.

When using commercial support services, time is often spent trying to reproduce the problem with a minimal system (perhaps by writing some throw-away test code) in order to demonstrate the problem to the support vendor. This can sometimes be very difficult to do, especially when developing code for specific hardware which the support vendor does not have. Instead, it is often quicker to use the Internet to look for similar problems.

Best practice in companies using Embedded Linux is to actively encourage their developers to use resources available on the Internet to help solve problems or find the best way to implement something. This is often done anonymously using web e-mail accounts to avoid competitors finding out about the development activities of the company. There is far more to be gained by interacting with a pool of experts in the community than there is to lose by hiding the work that might be available to all as a result.

Testing

With the exception of low-level, hardware device drivers, most software that is written for Embedded Linux hardware can be run with few changes on a standard Linux PC. Such software can be written and tested before hardware is available. Yet few Embedded Linux projects take advantage of this. Linux hosts have more tools available for application development and testing, e.g. valgrind. Also, the edit / compile / debug / test cycle is much shorter when cross-compiling environments aren't used.

For testing on the target hardware, Linux's shell and other scripting utilities allow tests to be executed. Device drivers are often tested using small test applications invoked from the shell

[4] On a typical Linux workstation, only a process with root privileges is able to create a real-time thread because it has the potential to hang the system.

prompt. If the target hardware has an ethernet interface, numerous mechanisms are available with which automated tests can be run on the target, driven by scripts running on a host PC. Automated regression tests may be implemented using tools such as Tcltest, driving tests remotely from a host PC. Tcltest provides mechanisms for scripting tests and logging results, yielding pass/fail status for each test.

For Embedded Linux software development, best practice is to test as much as possible in a Linux host simulation environment. For testing on the target hardware, small command line applications or RPC facilities are then used to exercise and examine the software running on actual hardware. Tests should be automated as much as possible.

GPL Licencing

The GNU Public Licence (GPL) is often sited as an issue with regard to using Embedded Linux in commercial products. Closed source proponents suggest any company that uses GPL in its products risks a future lawsuit. This is scaremongering. However, some companies do fall foul of GPL simply as a result of ignorance; a few companies even choose to deliberately ignore GPL and hope no-one finds out, Failure to conform to GPL risks the product being forcibly withdrawn from the market.

What should you do to ensure you conform to GPL? Keep in mind that GPL exists to allow others to take your modifications of GPL code (which you got for free) and improve on them, perhaps adding features or perhaps fixing bugs. All GPL sources used in the product must be published to allow others to rebuild the GPL code and to potentially take your modifications and perhaps use them in other GPL software. But what is to stop a company publishing most of its GPL code while retaining some key changes for its own advantage? How does a company prove that the GPL sources it publishes accurately represent the GPL code in use in its products?

In order to prove that the published GPL sources exactly match those used in building a product's software, build configuration files (e.g. the Linux kernel's .config file) and cross compilers should also be included with the sources. In this way, a third party is able to compile the GPL code and verify that the same signature exists in the actual product. However, the situation is clouded for embedded products where the root filesystem is burned into flash using a proprietary and secure binary format. The effort required to crack open and reverse engineer the binary data in order to verify its signature can be prohibitive, though not impossible. The highly recommended website http://www.gpl-violations.org/ discusses these issues in much more detail.

It is all too easy to change GPL code without realising that it is GPL licenced, despite GPL licence text in comments at the top of each source file. One of the most common issues is actually tracking which GPL code has been included in the product's software and of that set, which components have been modified during a project. A bigger problem occurs when GPL code has been linked with proprietary code because under the terms of GPL, the proprietary code can become a *derived work* of the GPL code and so it itself has to be released under GPL. The exception to this is linking with code licenced under LGPL where only the changes made to the LGPL code need to be published. Project leaders must audit public domain code which is included in a product. Engineering GPL code out of a closed source application just before the product is due to ship is bad news!

Fortunately, a Linux build tool called buildroot (http://www.buildroot.org) simplifies the mechanics of providing a source release of all GPL components. Buildroot provides a unified environment for building open source software components into a root filesystem, automatically downloading source tarballs and applying patches for cross-compilation. The builder may add additional patches, thereby making local changes to the open source code. Since buildroot comes with makefiles and configuration scripts for cross compiling open source components, it is straightforward to wrap it up in a source file release for GPL.

Recommendations

Without exception, companies who make best use of Linux in embedded systems are those who embrace the things that Linux brings to the table. Linux is a paradigm shift in embedded software development. The table below summarises what companies should do to make maximum use of Linux in their embedded products.

	Recommendation
1	Choose your embedded hardware vendor carefully. The quality of Linux support provided by hardware vendors varies enormously. Don't assume that big corporations provide good Linux solutions. Be wary of custom Linux distributions where the vendor has no track record of feeding changes for their hardware into the community. In hardware designs, favour chip vendors who provide real Linux OS support, where kernel device drivers for their devices wouldn't look out of place if they were distributed with the standard kernel sources.
2	Identify real-time constraints early and prototype Linux solutions to determine if Linux can meet those constraints. Implement functions that have real-time constraints in the kernel.
3	Design for UNIX/Linux. Expect to redesign or rewrite significant parts of existing RTOS code if porting it to Linux.
4	Use the UNIX/Linux process model to your advantage. Subsystems which have well-defined interfaces yet are otherwise independently scheduled should be implemented as separate processes. Avoid threads unless NPTL is available. But even if NPTL is available, consider whether separate processes would work as well because several common library calls are not thread safe. Debugging multithreaded applications is also relatively difficult.
5	Identify and evaluate standard Linux subsystems early in the project. Don't be reluctant to throw away proprietary code when Linux provides the same functionality. In most cases, the Linux solution will work better than the proprietary code under Linux because it has been designed for UNIX/Linux, unlike most proprietary solutions.

Conclusions

It would be easy to conclude after reading about the things to consider when developing with Embedded Linux that Embedded Linux is not the way to go. In some cases, it might be true, for example in systems with verifiably hard, real-time constraints and an architecture where commercial, real-time variants of Linux such as RTlinux cannot be used. Established companies with products built with RTOS are coming under increasing pressure by competitors using Embedded Linux because they are unable to keep up with the features offered by the Linux-based products. Yet when established companies dip their toes into Embedded Linux, they can get burned badly. Failed Embedded Linux excursions are almost always the result of not accounting for the differences between RTOS and Linux as outlined in this document. Best practice for Embedded Linux projects is to design and develop software for embedded products in the same way as for regular Linux systems.

Readers are encouraged to post comments and contribute to this discussion topic using a Web Forum hosted at

http://www.katalix.com/forums

You will need to enter a secret code in order to register an account on the forum. The secret code is `nospamplease.`

About Katalix Systems

Katalix Systems Ltd was founded in 2004 with a mission to help companies make best use of Embedded Linux. Clients include companies developing products for networking, digital TV and off-the-shelf CPU board markets. Its founder, James Chapman has 18 years experience in embedded software product development for LAN/WAN networking, telecoms and digital TV products. He has worked with Embedded Linux for 8 years, having previously used commercial RTOS (VxWorks, Nucleus). He has contributed work to the Linux kernel, including several device drivers and board support code. He wrote and now maintains the OpenL2TP software package.

*** End of Document ***