Deploying Linux Embedded Systems

Vincent Sanders
Daniel Silverstone

Revision History

Revision 1.00	9th March 2009 Initial Release.	VRS, DJAS

Table of Contents

This article considers the issues surrounding deploying Linux-based embedded systems.

1. Introduction

This article is the sixth in a series demonstrating the fundamental aspects of constructing embedded systems.

We discuss the issues which arise when deploying embedded systems onto real systems.

The whole series assumes a basic understanding of a Linux-based operating system. While discussing concepts and general approaches these concepts are demonstrated with extensive practical examples. All the practical examples are based upon a Debian- or Ubuntu-based distribution.

2. What is being deployed?

The final stage of a project is to package all the software and hardware into something that can be delivered as a product. The techniques we have already presented of reproducible automated builds will enable a software release to be taken and packaged at any point.

The release process should be viewed as an important part of the project and not neglected; several projects have failed because their underlying software was adequate but the release was a failure.

Any release should be clearly identifiable generated with a branch within the RCS system or a copy of *all* the source files used to generate the binary output. This is another example where automation and reproducibility will produce superior results.

One issue which seems to be a perennial problem is that of hardware changes for production manufacture which require software changes. This is of course a change in project requirements after the software project is complete! Nonetheless this happens often. Strategies for dealing with this issue include

- Producing the software deliverables after the hardware has been completed. This approach is rarely acceptable as it introduces additional delay after a hardware platform is complete.

- Producing software flexible enough to deal with the hardware changes. This approach is reasonable to a certain degree. Unfortunately this means the software engineers must anticipate possible hardware changes and write code which is speculative in nature to cope. This code cannot be properly tested, requires time and effort to produce and may contain errors and security issues.

- Design hardware ready for manufacture from the outset. This is the lowest cost and most powerful option as it removes the entire “redesign for manufacture” step from a hardware process and ensures the hardware performs as the original designer envisaged. It also means the software platform can be stabilised along with the hardware which will be manufactured. Unfortunately this approach is used by few projects because there is an assumption it will cause additional expense.

There is always a human element in the release process which, due to the complexity of the task, is often overwhelmed. Automation can assist in this as can having a well documented procedure. Successful projects have often dry run a series of releases, to refine their process, before final deployment.

If the product can be updated the update procedure should be straightforward and *extremely* well tested. A good update system means a product might ship with numerous other issues which can be fixed later. This is not to be encouraged as a development model but does allow a solution if things have gone wrong.

Effective testing is important to a successful product. When a users interaction is limited with a system it is especially important that those interactions behave in a rational and reliable manner. We have observed products where even the most trivial testing cannot possibly have been performed; e.g. music players which crash playing the example music provided. Simple on or off lighting systems which do not always switch the lights on.

Automatic testing is useful but if not possible at least a basic checklist for manual testing would help.

3. Practical considerations of a deployment

Embedded systems by their nature often introduce additional issues which would not normally have to be considered.

3.1. Storage

The first of these issues is that of storage. So far in this series of articles all our examples have been compressed cpio archives retrieved from removable media or the network. The running systems have been completely volatile and no state is retained across boots. These systems might suffice for trivial control or monitoring applications but have the drawbacks of relatively long load and boot times and no way to configure their behaviour.

One approach might be to use a standard hard drive and boot from a file system on it. This is the way most full operating systems start, is well tested and relatively reliable. This solution is completely impractical for most embedded systems where moving media is impractical. Spinning media has the drawbacks of power usage, noise, mechanical failure and sensitivity to shock. Because of these issues embedded systems often resort to solid-state flash.

Unfortunately flash media does have some issues of its own. The primary problem with all current flash devices is the number of times they can be written to. A flash device is generally split up into a number of sections, the size of these sections varies between a few hundred bytes and hundreds of kilobytes depending on the device and the technology used to implement it. Each section may be individually erased (where all bits are set). When a device is programmed, bits are cleared to achieve the desired data pattern. Because of their physical structure all flash media has a maximum number of time a section may be erased. The failure manifests as the erase being unable to set all bits back to one or bits other than those cleared becoming zero during operation.

Current flash devices can be split into three categories (other devices exist but are far less common) there are:

NOR
: NOR based devices are relatively small (largest individual device is around 16Mb) are slow to erase, program and read from but every erase block is guaranteed to be good at manufacture (no bad blocks). NOR devices also have the property they can be directly accessed as a memory which makes their use common as boot devices. Their usage is declining in favour of other technologies.

NAND — NAND memories are faster and larger (current largest single device is around a few Gigabytes) than NOR but they are not random access and data must be read and written in pages. The erase block size generally contains many pages of data which increases the complexity of software to manage data. These memories may be manufactured with bad blocks (only block 0 is guaranteed) and blocks are expected to fail during usage. NAND memories generally provide additional Out-of-Band (OOB) data which is used to mark pages as bad and to provide some error detection and correction.

Block interfaced — Strictly speaking this is not a flash technology, only a method of access to the other two types of flash (almost certainly NAND). This type of flash is found in all USB, CF and MMC type devices. The flash is presented as a linear array of 512byte blocks as on a hard disc. The underlying flash technology is completely abstracted away by a controller within the device. There is no indication of the actual layout of blocks on the flash device, the strategy being employed to map logical blocks to physical flash or if any distribution of blocks to reduce erase cycles is being performed. Often there is also no protection for the internal logic of these devices loosing data and state during unexpected conditions such as loss of power which may cause corruption or in some cases the device may cease to function altogether.

The selection of the correct flash memory is an involved and complicated process and many factors must be considered. The final decision is of course project dependant. The most common solution currently in use on non-PC systems is a NAND device as a boot media containing the boot-loader, the boot kernel and modules and operating system. A second piece of removable “block interfaced” media (MMC or SD is popular) is used to store configuration and user data. This approach means the core system can be stored on flash which, with a proper selection of file system, will be stable and reliable with few erase cycles excepting OS upgrades. The frequently changing user data is stored on the less reliable but replaceable device.

An area of continuous debate within the Linux embedded community is that of file system selection for flash devices. Every file system has a differing pattern of reads and writes depending on its intended use. For example the FAT file systems write to their allocation tables over and over, without suitable software strategies the underlying flash erase block would soon exceed its maximum erase cycle count.

Within Linux non-“block interfaced” devices are accessed using the MTD subsystem. The MTD system presents a unified API to both NOR and NAND devices and provides numerous hardware device drivers for many platforms. The MTD API however requires file systems specifically written to that API to be effective. The primary file system in general use is JFFS2 which provides a fully journaled fault tolerant wear levelling file system. The primary drawback to the file system is it can take several seconds to mount large devices. This may be significantly reduced, at the expense of some storage capacity, by using summary records when constructing the file system. In addition sensible partitioning of the flash into smaller pieces reduces mount time still further.

The file system of choice for “block interfaced” devices is far less clear. The controllers in these devices vary so widely and are tuned for so many differing loads it is difficult to make generalisations. The main concern when dealing with such devices is that writes should be minimised and, if possible, grouped into a small number of large writes instead of many smaller writes. For example the ext3 file system is possibly a pathological worst case for a great number of commercially available compact flash cards. One project attempted to use this file system for a root file system, the cards became inoperable after only a dozen or so reboots.

The main concerns about selecting storage devices and file systems have been touched upon here. Many references are available elsewhere on this subject and should be consulted to gain a more complete understanding.

3.2. Boot-loader

One area sometimes overlooked in embedded systems is that of the boot-loader. For a PC this is hardly ever a consideration, the BIOS starts a loader like GRUB and that starts the kernel. On non-PC targets the situation is somewhat more involved. Some architectures have a specific loader such as Open Firmware whereas others like MIPS and ARM have a selection.

Selecting a suitable boot-loader is often a challenge, however there are often several to choose from. On the ARM architecture for instance, the open source U-Boot and the proprietary ABLE loader are popular. If your are customising an existing solution your provider might already supply a loader pre-compiled and installed.

Loaders, just like operating systems, come with various different features and options and the loader suitable for development might not be suitable for deployment. In contrast to this the selection of the loader should not delay a project or greatly influence it, the loader after all is supposed to perform the job of getting the operating system started as quickly as possible, all other features are extra.

One aberration with boot-loaders is the continuing desire, by software engineers, to re-implement new ones. This desire should be resisted wherever possible as it adds no direct value to a project and inevitably leads to huge uses of resources to solve low level issues. To give an idea, the ABLE proprietary loader has in excess of a decade of engineering resources expended on it.

3.3. Production

Production is the overall term for the manufacture of the hardware, programming of firmware and software onto the hardware and factory testing. This element of a project is typically handled by a third party manufacturer. All that is required in these cases is a clear set of programming files and clear instructions to the manufacturer.

Some projects may require local programming and test before dispatch, in this case automation and verification is critical to avoid costly recalls.

3.4. Software licencing

The authors are not lawyers so the advice in this section must be seen only as guidance. All legal matters must be referred to a qualified professional.

The operating system produced and deployed using Open Source software will have some licencing terms to comply with. This need not be arduous and is a small price to pay for the huge quantity of software which can be used without fees.

A general guide is to provide all the sources used to build a system in the same way you received them and a separate archive containing all the changes and configuration made. This source can be distributed on the install media or via a website but the user should be informed within the packaging the sources are available and how to retrieve them.

You do *not* have to distribute the sources to your proprietary programs (provided you are not required to by any libraries it links to). A sign of a good project is that the distributed components, both open source and binary, can be used by a suitable proficient user to rebuild the system as they see fit.

4. Getting help

Although we have tried to educate throughout and make the reader self sufficient it is sometimes necessary to seek out expert help. This can sometimes be acquired through the open source communities.

Please remember, when addressing open source groups, that they give freely of their time and although you may feel your issue is of utmost importance, they may not share your view. Ensure you have thoroughly searched all the FAQs and documentation before posting queries to forums or mailing lists as a repetition of a question which has already been addressed several times may result in a negative reply.

Embedded projects because of their nature seem to produce developers who are keen to solve the issue for their single project and move on. This does not fit terribly well with the open source ethos of giving back to the communities software is taken from. Submitting good bug reports and useful modifications to a project need not be a time consuming activity and increases the likelihood of useful assistance. Sending demanding or derogatory messages to developers is likely to get your message filed in a similar manner to spam.

One group of developers which warrants special mention are the Linux kernel developers. There are many of them, they develop code of the highest quality and more importantly they maintain that code over a long period. They are also very busy people, you should endeavour to make their lives easier.

Your project will require a kernel, it may require the development of new hardware drivers. These drivers must be submitted to the relevant kernel maintainer using the correct mailing list, it is unlikely a kernel maintainer will search out the driver for your device.

A driver in the mainline kernel is maintained alongside every other driver which means the support burden is vastly reduced and the driver is automatically moved forward. The next project that requires use of that driver will be able to use it without further development.

To be included a driver must be of a certain quality. If the driver is developed correctly this should not become an issue. Drivers should use the generic kernel API and follow the coding standards clearly laid out in the kernel documentation. Developing kernel drivers is outside the scope of this article but numerous good references exist the one we generally recommend is Linux Device Drivers [http://oreilly.com/catalog/9780596005900/].

Sometimes you need assistance with a project on your terms and not those of the open source communities. When this occurs there are a number of reputable companies to turn to, many who employ open source developers.

5. Wrapping up

This series of articles outlines the basic method and ideas necessary to produce Linux kernel-based embedded systems. We have attempted to highlight the common mistakes and issues experienced by many when embarking on embedded projects.

The main points made have been:

- Have a clear project goal,
- Automate everything possible,
- Ensure repeatability,
- Use existing tools,
- Customise existing solutions.

We hope this series has been at least partially enlightening. There are of course many more areas not touched upon in this series which we hope to cover in future articles.

6. About the authors

Vincent Sanders	Vincent is the senior software engineer at Simtec Electronics and has extensive experience in the computer industry. He has worked on projects from large fault tolerant systems through accounting software to right down to programmable logic systems. He is an active developer for numerous open source projects including the Linux kernel and is also a Debian developer.
Daniel Silverstone	Daniel is a software engineer at Simtec Electronics and has experience in architecting robust systems. He develops software for a large number of Open source projects, contributes to the Linux kernel and is both an Ubuntu and Debian developer.
Simtec Electronics [http://www.simtec.co.uk]	Simtec is a full solutions provider with a proven track record of helping clients with all aspects of a project, from initial concept and design through to manufacturing finished product. With 20 years in the industry, and producing ARM CPU modules since 1992, Simtec's wide experience in embedded systems and the Linux kernel provide a strong base on which to build custom hardware and software solutions, from the smallest of USB devices to the largest complex Linux systems. Simtec's custom-off-the-shelf design service, utilising a range of pre-designed modules of various functions, allows for rapid design and prototype turnaround, reducing time-to-market. Simtec also provide a full software development consultancy with an extensive range of products from boot loaders to full Linux based operating system environments and a range of development boards showcasing Simtec's modular designs.