Software updates for embedded Linux: requirement and reality

Chris Simmonds

Guest Contributor

Software updates in embedded systems is an often neglected feature, and yet it is crucial to the success of the “Internet of Things”. In this white paper, we will look at the issues surrounding software updates and outline some of the things to consider when implementing an update mechanism.

Why are updates a requirement?

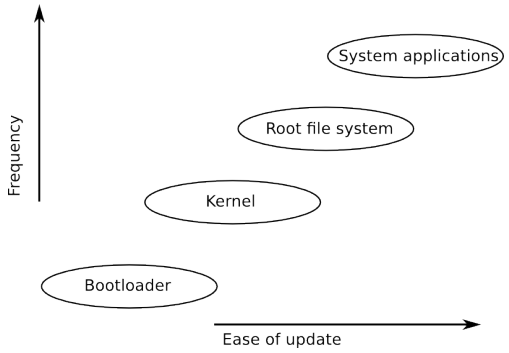
Embedded devices are becoming ever more complex and simultaneously becoming ever more interconnected. The first makes it very likely that updates will be needed to fix problems as they arise, and the latter opens up the possibility that updates can be pushed over the network. Adding the software update mechanism to your product provides the ability to:

- Deploy security patches
- Fix bugs
- Enable or deploy new features

On the other hand, devices with static software – those that cannot be updated in the field – are open to attack, with consequent risk to end users and negative publicity for the vendors and developers. It is safe to say that software updates for devices in the field is truly a requirement.

Requirements of an updater

Updating a device remotely is a tricky proposition:



lots of things can go wrong. The problem was well stated by Gilad Ben-Yossef [1] in his paper titled “Building Murphy-compatible embedded Linux systems”. Let’s begin by considering the basic requirements of an automatic, remote update mechanism. It must be:

- Secure, to prevent the device from being hijacked
- Robust, so that an update does not render the device unusable
- Atomic, meaning that an update must be installed completely or not at all
- Fail-safe, so that there is a fall back mode if all else fails
- Managed remotely, to enable unattended updates
- Auditable, so that you can tell what updates have been applied
- Able to preserve user data

A good update mechanism has to meet all of these to be effective.

What to update?

Embedded Linux devices are very diverse in their design and implementation. However, they all have these basic components:

- Bootloader
- Kernel
- Root filesystem
- System applications
- Device-specific data

Some components are harder to update than others.

Let's look at each component in turn.

Bootloader

The bootloader is the first piece of code to run when the processor is powered up. The way the processor locates the bootloader is very device-specific, but in most cases there is only one such location, and so here can only be one bootloader. If there is no backup, updating the bootloader is risky: what happens if the system powers down midway? Consequently, it makes sense to leave the bootloader out of the update strategy. This is not a great problem because the bootloader only runs for a short time at power-on and is not normally a great source of runtime bugs.

Kernel

The Linux kernel is a critical component that will certainly need to be updated from time to time. There are several parts to the kernel:

- A binary image that is loaded by the bootloader
- Many devices also have a Device Tree Binary (DTB) that describes hardware to the kernel, and thus has to be updated in tandem. The DTB is loaded by the bootloader
- Kernel modules, stored in the root filesystem

The kernel and DTB may be stored in the root filesystem, as long as the bootloader has the ability to read that filesystem format. They may also be in a dedicated partition. In either case, it is possible to have redundant copies.

Root filesystem

The root filesystem contains the essential system libraries, utilities, and scripts needed to make the system work. It is very desirable to be able to replace and upgrade all of these. The mechanism depends on the implementation. Common choices for embedded root filesystems are:

- Ramdisk, which is a binary image that is loaded from a partition or filesystem by the bootloader at boot time. To update it, you just need to overwrite the ramdisk image and reboot
- Read-only compressed filesystems, such as squashfs, stored in a partition. Since these types of filesystems do not implement a write function, the only way to update them is to write a complete filesystem image to the partition
- "Normal" filesystem types. For raw flash memory, JFFS2, and UBIFS formats are common, and for managed flash memory (eMMC and SD cards) the format is likely to be one of ext4, F2FS, or btrfs. Since these are writeable at runtime they can be updated file-by-file, or you can write a complete filesystem image in the same way as for compressed read-only filesystems

System applications

The system applications are the main payload of the device; they implement its primary function. As such, they are likely to be updated frequently to fix bugs and to add features. They may be bundled with the root filesystem, but it is also common for them to be placed in a separate filesystem to make updating easier and to maintain separation between the system files, which are open source, and the application files which are often proprietary.

Device-specific data

This is the combination of files that are modified at runtime and includes things like configuration settings, logs, user-supplied data, and the like. It is not often that they need to be updated, but they do need to be preserved during an update. Such data needs to be stored in a partition of its own.

Components that need to be updated

In summary, an update may include new versions of the kernel, root filesystem, and system application. The device will have other partitions that should not be disturbed by an update, as is the case with the device runtime data.

How should the update be applied?

Embedded devices almost universally use flash memory for storage. Flash memory can be divided into partitions in a way similar to hard disks. Components such as the kernel, DTB, and ramdisk may be each stored in a separate partition, but it is more common for them all to be stored together in the root filesystem. A filesystem has a format such as UBIFS or ext4, and is contained within a partition. So, it follows that you can apply updates at several different levels: file, package, or partition image. There are pros and cons for each.

File-by-file

Updating file-by-file is the most obvious approach, and indeed many projects have a homegrown updater based on tar balls and scripts that will apply updates file-by-file. They almost always fail to fulfill some of the criteria given above. Atomicity is the first big stumbling block. It requires careful coding to make sure that individual file updates are atomic. For those interested in the details, the technique is to write the new content to a temporary file and then use the POSIX rename (2) function to replace the old file with the new one. The rename function guarantees that this is atomic so long as both are in the same partition and the filesystem is POSIX compatible (which excludes FAT filesystems) [2]. However, even when you have solved this problem you will encounter the bigger problem of ensuring atomicity over a group of file updates, which requires knowledge of the dependencies between them and how to roll back a failed partial update. This is a hard problem and it can be solved somewhat by using a package manager.

Package

Desktops and servers have well-established Linux distributions with package managers, such as the Red Hat Package Manager, RPM, and the Debian package manager, dpkg. Do they work for updating the filesystems of an embedded Linux device?

Package managers store enough state to be able to detect a failed update and rollback to a previous version. They can manage dependencies between packages. They can even do updates on a live system, so it is only necessary to reboot if the kernel or its modules have been changed. They keep an audit trail of what was installed and when. Thus package managers meet at least some of the requirements of an updater and are always a better option than a homegrown updater.

But an embedded device differs from a server sitting in a data center. There is a much greater chance of a failed update due to loss of power or loss of network connectivity, and package managers cannot guarantee recovery in all such cases. Which means that there is a chance that a device will not boot up following an update. In a data center, as a last resort you can go to the machine and complete the update manually. But embedded devices are often in remote locations or are difficult to access for a variety of reasons, making manual intervention expensive. In the case of user-facing devices, it may be possible for the user to remedy the situation, but at the cost of inconvenience to the user and loss of confidence in the device.

A subtler problem has to do with the combination of versions of software that package update allows. Over a period of time a population of devices can experience version drift because of failed or rolled back updates on some devices, changing update schedules, and even manual updates where that is

possible. You cannot say absolutely that the entire fleet of devices are running exactly the same versions of all software.

So, package managers are a good choice where there are frequent small updates to a system, where occasional update failures are acceptable, and where you are able to perform the additional QA required to ensure a spread of versions of software can coexist.

Image

Updating a whole partition containing a filesystem image or kernel binary fits in well with the way most embedded Linux distributions are put together. The output of a build system such as the Yocto Project or Buildroot is ultimately a number of images ready to be installed in flash memory on the target device.

Image update has the advantage that you can be sure that all devices in the field have exactly the same software build. Applying the update is fairly simple and so the implementation of the update agent can be quite simple as well. This means that it is easy to validate that it works and is robust.

But because you are updating whole partitions rather than files, the size of the update is usually larger than a package-based system. Also remember that it is not possible to update a live filesystem image, meaning that the update has to be written to an alternative partition and the system has to reboot before the update can take effect.

Making updates robust: Murphy-proofing

The great worry about remote automatic update is that some devices will not operate correctly afterwards, remembering that Murphy's law states that if it can go wrong, it will go wrong eventually. There are two important features that are needed to recover from this situation: rollback and watchdog.

Some failure modes can be detected before booting into an updated system. The integrity of the updated image data can be verified by a hash code, and the veracity of the source of the update by checking the digital signature. These checks can be done by the bootloader, which can choose to boot a previous, known working, system image instead if it finds a problem. As we have noted already, the bootloader itself is not a candidate for updating, and so it is the natural choice for these tests.

But what if the update seems fine but has an unforeseen bug that causes a system hang? This is where you need the belt-and-braces solution of a hardware watchdog. Most embedded chips have such a thing built-in. If it is enabled by the bootloader it will force a system reset if it is not triggered at regular intervals thereafter. In the

case of a system hang, the watchdog times out and the system reboots. Back in the bootloader code, it is possible to discover that the reboot was caused by the watchdog and so to rollback to the last known good version.

Security

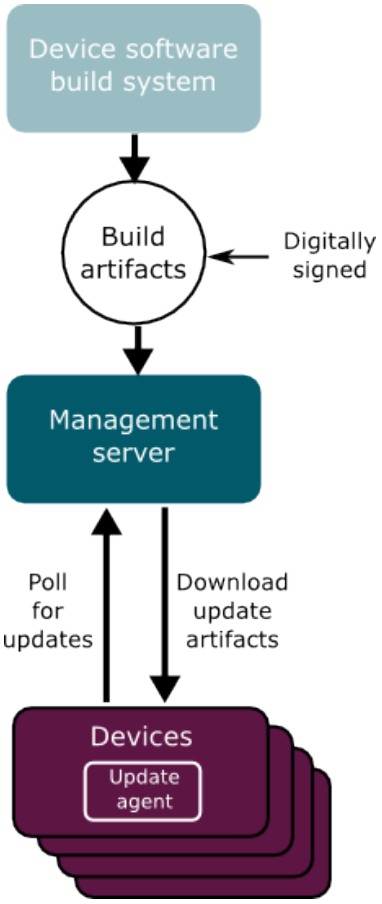
Security is the next big concern. By providing a mechanism to load new software onto your device, you are opening up the possibility that someone else can do the same and thereby hijack it. Any update method must provide a means of verifying updates are from a valid source. In practice, this can be achieved using digital signatures: the update is signed by you, the provider, and checked on the device before installing it, and again in the bootloader before loading an image.

You may want to make the update channel secure as well, using an encrypted protocol such as https, so that you can be sure that the update byte stream is not visible to the outside world and cannot be tampered with.

Putting it all together

It is probably clear by now that there is more to an automatic over-the-air update system than just the update agent running on the device.

The following diagram shows the main components:



The process begins with the device build system, such as the Yocto Project. It generates the images for the device, which include the kernel, root filesystem and applications filesystem. The update agent is packaged into the root filesystem. These images are used to create an update package in the format required by the update agent, which would include a manifest of some sort, giving the hardware revision that this update is targeted at, expected minimum revision level of the existing device software and revision level of the enclosed build. The package is signed so that it can be authenticated later on.

The update server manages the deployment of update packages. It is a central point for deploying updates to a population of devices and performs such tasks as monitoring the current software build installed on each device and schedules the rollout of new releases.

Finally, the update agent polls the server to discover if a new version is available and also feeds back information about the current state of the device. When there is a new version, it will download and install it.

Updates in the real world

The whole area of remote update is only just getting attention from system designers. At the moment, there are some homegrown systems out there, most with some or all of the problems I have outlined, plus the overhead of developing and maintaining an internally developed system. But now we are entering a phase where open source and proprietary systems are becoming available. Open source solutions such as Mender (www.mender.io) are attractive not only because the implementation is transparent and verifiable, but also because they are easy to integrate and adapt as needed. And its permissive Apache 2.0 license provides the flexibility organizations need in their software stack.

References

[1] Gilad Ben-Yossef, Building Murphy-compatible embedded Linux systems, https://www.kernel.org/doc/ols/2005/ols2005v1-pages-21-36.pdf

[2] The rename (2) function: http://man7.org/linux/man-pages/man2/rename.2.html

Chris Simmonds *is a freelance consultant and trainer who has been using Linux in embedded systems for over 15 years and teaching others how to do so for almost as long. He is the author of the book "Mastering Embedded Linux Programming" and is a frequent presenter at open source and embedded conferences, including Embedded Linux Conference, Embedded World and Android Builder's Summit. He has been running training courses and workshops in embedded Linux since 2002 and has delivered hundreds of sessions to many well-known companies.*