# INTRODUCTION: A TYPICAL EMBEDDED SYSTEM

**It's not always clear what separates ordinary Linux from embedded Linux. This article takes a look at the parts that make up a typical embedded system, starting with the bootloader and ending with end-user applications.** **JOHAN THELIN**

The very first step in starting an embedded Linux system does not involve Linux at all. Instead, the processor is reset and starts executing code from a given location. This location contains a bootloader that initializes the device and sets up the basic necessities. When everything has been prepared, the Linux kernel is loaded and started. The kernel then initializes all the devices before mounting the filesystems and starting the userspace applications.

The Linux kernel and userspace are not merely a simple blob that is loaded and run. The kernel consists of a system-specific configuration and usually some tweaked initialization code. The userspace holds software libraries, data and several applications, all interacting to form a system. Each of these components is handpicked for the task and device in question in order to get a compact and well-performing system. Figure 1 shows the basic sequence of events.

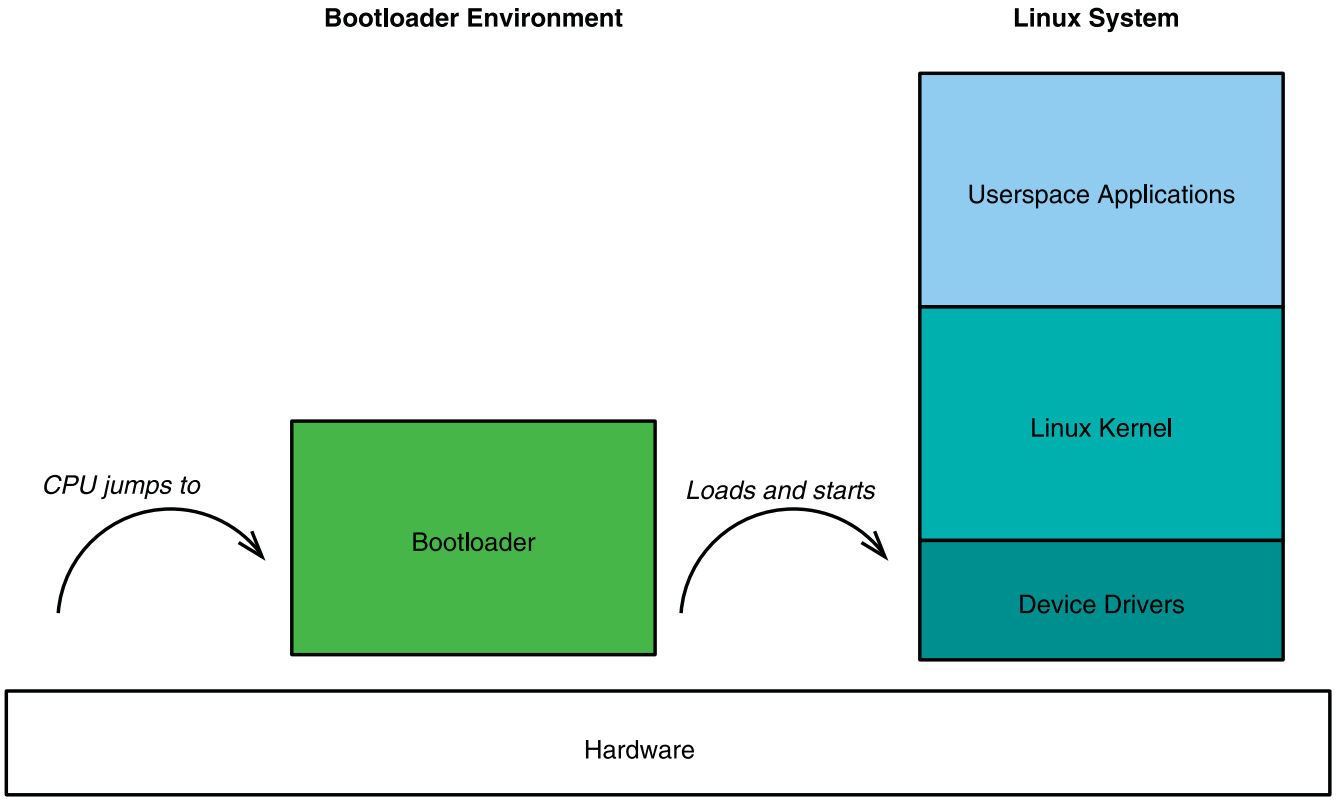


Figure 1. An Embedded Linux System Booting

## THE BOOTLOADER

The bootloader is among the first pieces of software to run on the system. It basically has two tasks: initialize the system and load the kernel. The initialization can be to set up a UART to be used as a serial debug console and to configure the system's memory controller. For instance, if your system is using an SDRAM, you probably will have to set up the controller with regard to the memory's physical features. This includes page sizes, the number of columns, supported read and write widths, latencies and so on. In these days of portable devices, there is usually a plethora of settings for saving power when it comes to memory.

In addition to the basic tasks required by the bootloader, it is typical to provide some sort of command prompt where common low-level tasks can be carried out. These tasks usually include peeking and poking at random memory addresses, downloading and storing a Linux kernel image in Flash and setting bootargs for the kernel to interpret.

Examples of common bootloaders for embedded systems are Das U-Boot and RedBoot. Both support the basic tasks—meaning they can manage Flash, networking and serial communication. They also are available for several processor platforms, such as x86, ARM, PowerPC and more. You can add your own commands to both of them as well. This makes it possible to debug custom hardware without involving Linux, reducing the complexity of the system during the testing phase.

## THE LINUX KERNEL

The kernel itself is not very different from an ordinary desktop kernel. However, there are two major differences. First is the initialization, which often is system-specific. Second is that you probably know exactly what hardware will be used, so you can include all the drivers as part of the kernel and avoid the need for modules (unless you have proprietary drivers, of course).

When starting a desktop or a server system, the common scenario is that the kernel probes for hardware and loads the corresponding drivers as modules. This makes it possible to add hardware and still have a working system. You also can add drivers for new hardware without having to recompile the entire kernel. On an embedded system, you can optimize boot time by including all drivers in the kernel, but also by hard-coding parts of the available hardware, avoiding the need to probe for all devices and settings.

Returning to the standard PC, each machine starts and looks about the same during initialization. In the embedded case, each piece of hardware is unique, and you generally have to initialize the custom hardware. This means you actually will have to write code to set up your kernel for your board, which is usually easier than you think. For starters, lots of boards already are supported in the Linux kernel, and you usually can choose one of those as a starting point. Second, there are drivers for the most common peripherals, and again, you typically can find a good starting point, even when you have to create something of your own. So, the process is more or less to study the data sheets for your board and express what you learn to the kernel (something that can be both intimidating and daunting).

Embedded systems often are more limited than your average computer when it comes to system resources, so it is important to keep your kernel's footprint small. That, in turn, makes the kernel configuration stage important. By limiting configuration to a minimum, you can save those extra bytes needed to fit everything in.

## THE C LIBRARY

The standard C library is one of the key components of any Linux system. It provides the userspace applications with a predefined interface, making them portable across different versions of the Linux kernel, as well as between different UNIX dialects. It basically acts as a bridge between the userspace applications and the kernel.

## FILESYSTEMS

Choosing a filesystem for your embedded system depends on many factors. Do you need to be able to write to it? Do you value size or speed? Do you want to be able to replace the filesystem without replacing the kernel?

You also need to be aware of your storage medium's limitations. For instance, Flash has a limitation when it comes to how many times each cell can be written. To prolong the life of a Flash-based device, it's a good idea to use a filesystem that has been adapted for this purpose.

There are numerous filesystems from which to choose, but the following three are interesting as they show some important factors you should take into consideration:

- initramfs: a filesystem that is embedded into the kernel image. If your kernel is compressed, the initramfs filesystem is decompressed alongside the kernel. This gives the system a performance advantage. The filesystem is kept in RAM as the device operates and can be modified. However, all modifications are lost upon reboot.
- cramfs/squashfs: two compressed read-only filesystems. Both of these systems let you create a compressed image that you can mount at runtime. The filesystem can be replaced without touching the kernel.
- jffs2/ubifs: compressed filesystems tuned for Flash devices. These filesystems can be written to permanently, and they try to minimize the "wear and tear" of the Flash blocks by spreading write operations across the device.

Luckily, you do not have to pick one of these filesystems; instead, you can mix them—for instance, starting from an initramfs image with the most basic tools and then mounting a jffs2 Flash partition for storing user data. As Linux allows you to mount filesystems into any location in your directory tree, you can make this transparent to the applications using the filesystem.

## CROSS COMPILATION

One of the interesting aspects of embedded development is that you are likely to encounter new processor families. Most of you have x86 hardware at home; some might have a SPARC, 68k or a MIPS system lying around. With embedded systems, you are likely to run into ARM, SH, PowerPC or MIPS, among others.

The implication of this is that you must cross compile everything from your desktop build machine (your host machine) for your target device. The resulting binaries cannot be run directly on your desktop machine. You can do it using emulators such as QEMU that allow you to emulate common CPUs, but you will have to do some testing and probably some debugging on your target device.

Sometimes you can get a cross compiler from a vendor or distribution. You also can build your own. Building your own cross compiler used to be a real pain, but these days, you can use crosstool from Dan Kegel. Crosstool is a set of scripts and patches that allows you to build gcc and standard libraries for your platform of choice.

Crosstool's greatest feature is that you can (attempt to) build any combination of compiler and standard library. This makes it easy to try to build a toolchain for an existing device.

## DISTRIBUTIONS and FRAMEWORKS

As fun as it is to roll your own, sometimes time does not permit it. A number of commercial players exist in the embedded Linux field, and many freely developed tools for building a complete embeddable environment also are available. The following list contains a few tools you might consider using:

- Buildroot: a set of Makefiles and patches for building a complete embeddable system. It generates everything from the cross compiler, the kernel and software libraries, and the userspace applications. The resulting system uses uClibc.
- Ångström distribution: another build framework for building embedded Linux systems. It also sports a package manager. This makes it possible to add and remove applications from the device directly, instead of having to build and download an entire system image or copy the application's files to the right locations manually.
- ScratchBox: a build framework for making embedded Linux application development easier. It has gained adoption through the Maemo development platform (the Nokia N7/8/9xx Internet tablets). It supports cross compiling entire distributions, can switch between glibc and uClibc and uses QEMU emulation of targets.

In all of those cases, it takes a bit of work to get the distributions running on a new system. As always with embedded systems, nothing is standardized, and size usually matters, so a bit of tweaking is more or less inevitable. However, having a framework for building a working system can be a real time-saver.

The version of the C library you usually find on your desktop machine is the GNU C library, glibc. It is a full-fledged C library, and, thus, a very large piece of software. For embedded systems, a few smaller alternatives are available: uClibc, newlib, dietlibc and others. These libraries try to implement the most commonly used interfaces in a minimalist way. This means they are mostly compatible with glibc, but not fully.

So, what does the C library contain that can be removed? uClibc, for example, skips the database library, limits the number of authentication methods that are supported, does not fully implement locale support, limits the math library mostly to doubles and leaves out some encryption functions. In addition, the kernel's structures are used directly whenever possible. Those and other things significantly reduce the size of the library.

What does this mean to you as an embedded developer? Most important, it means you can save quite a bit of memory, although you do so at the cost of compatibility. For instance, the decision to use the kernel's structures when applicable means the stat structure is different from the one used by glibc. You also have to limit yourself to flat password files and shared password files, unless you want to add a third-party library to handle authentication. More limitations exist, but generally speaking, most software compiles happily without patching.

### BUSYBOX

When you have a bootloader, a kernel and a standard library, the next thing on the wish list usually is a command prompt. One of the big stars in the embedded Linux world is BusyBox. The idea behind the project is that most standard applications, such as ls, cd, mkdir, ping and so on, share a lot of code. Compiling each program separately means that code handling things such as command-line arguments is repeated in each application. BusyBox solves this problem by providing a single program, busybox, that can handle all the tasks provided by all the standard applications. By creating symbolic links for all the individual commands and pointing them to BusyBox, the user can still enter the expected commands and get the expected results.

As with everything else in the embedded world, tuning and tweaking is important. When it comes to BusyBox,

you can handpick which commands to include, and for some commands, you even can handpick which command-line arguments are supported. If you don't need a particular command, simply don't include it in BusyBox. For instance, why keep ifconfig if you don't have a network?

When building a dynamically linked, default configured BusyBox on a desktop PC, it results in a binary that is just less than 700KB. This binary represents more than 200 commands and occupies more than 6MB of disk space on my Kubuntu-based system.

## ADDING MORE

Once you have all the key components in place, you can start building and populating a root filesystem. This involves adding BusyBox, device files and expected directories. You also might want to add /etc/password and /etc/shadow, init scripts and so on. All this is necessary, but to get your device to do something, you need to add your own applications.

When developing for embedded devices, you might find yourself in a system completely without a graphical interface. This usually means implementing your functionality as some sort of server. As more and more devices are networked, a Web server often takes the place of a user interface. Because Apache is a large piece of software, a common solution is to use a lightweight server, such as Boa, for configuration and information.

If you happen to have a display, you likely will want to put graphics on it. An X sever might sound like a solution, but the two most common toolkits for building graphical interfaces, Qt and GTK+, also support using the framebuffer directly—again, saving both memory and computing resources.

And, that is what engineering embedded devices is all about: making the most with as little as possible. Being able to fit the coolest features into a small system means bringing an attractive device, at a good price, to consumers. Using embedded Linux to do that means you can get done more quickly, cheaply and be more hackable than with a closed-source system.■

**Johan Thelin has worked with software development since 1995 and Qt since 2000. Having seen server-side enterprise software, desktop applications and Web solutions, he now works as a consultant focusing on embedded systems. He can be contacted at johan@thelins.se.**

## Resources

Crosstool: **www.kegel.com/crosstool**

Das U-Boot: **www.denx.de/wiki/U-Boot**

RedBoot: **www.sourceware.org/redboot**

uClibc: **www.uclibc.org**

newlib: **www.sourceware.org/newlib**

dietlibc: **www.fefe.de/dietlibc**

Buildroot: **buildroot.uclibc.org**

Ångström Distribution: **www.angstrom-distribution.org**

ScratchBox: **www.scratchbox.org**

BusyBox: **www.busybox.net**

Boa: **www.boa.org**

Qt: **qt.nokia.com**

GTK+: **www.gtk.org**