Guidelines for Choosing Bootloaders for Embedded Systems

By Justin Treon
Micron Software Engineer

The selection of bootloaders for embedded systems has expanded since the late 1990s; not just in the number of bootloaders on the market, but also in the number of platforms and operating systems supported by each bootloader. This article stands as a guide to selecting a bootloader for embedded systems to aid in reaching time-to-market goals by reducing redundant system, debugging, and feature enablement. Emphasis in the article is placed on Linux-compatible bootloaders. However, since viable bootloaders can boot many operating systems—from the most basic real-time operating system (RTOS)\(^1\) to a Linux server—this guide is relevant for any embedded system.

Bootloader Basics

The bootloader combines the concept of bootstrapping\(^2\) the system and loading the operating system’s kernel into memory with a loader, hence the name “bootloader.” The concept of using a small set of code to enable the system on which it runs is based on the old U.S. proverb, “to pull one’s self over a fence by one’s bootstraps,” which is an impossible task\(^3\). The minimal set of bootstrapping code required to initialize the system is between 50 to 150 lines of assembly, depending on the architecture and chipset. Despite the small code set, the size of modern bootloaders has swelled to more than 256KB to increase their portability across multiple boards and architectures. The feature set has also grown and bootloaders now include debugging features and the ability to select from multiple images to load into memory.

A full-featured bootloader is not required to boot a system, but strictly speaking, no bootloader is required if the bootstrapping code is supplied. In systems with NOR\(^4\) Flash or other directly-accessible memory, such as phase change memory (PCM\(^5\)), the bootstrapping code may be added to the beginning of an image that can be accessed directly from the nonvolatile memory. To do this, the code must be executed in place (XIP\(^6\)) directly from the nonvolatile memory. Systems with memory that is not directly accessible (NAND\(^7\) Flash) often requires additional bootstrapping code to drive the NAND controller,

---

\(^1\) An RTOS is applicable for real-time applications. It enables the execution of real-time events. Examples include Accelerated Technology’s Nucleus\(^8\) and Wind River’s VxWorks\(^8\).

\(^2\) Bootstrapping is the initial boot sequence for minimum functionality, with emphasis on physical memory configurations, memory timings, and bus timings.


\(^4\) NOR is a nonvolatile storage technology consisting of the parallel alignment of the transistors. It uses hot electron injection for programming. In comparison to NAND, the architecture enables faster random reads and slower block writes. NOR is best used for code storage.

\(^5\) PCM is a bit-alterable nonvolatile storage media designed to replace NOR and NAND. Rather than storing a charge in a floating cell as in Flash, cells are heated by passage current to form a crystalline or amorphous state to store the bit value.

\(^6\) XIP code is directly accessed in nonvolatile memory without a shadowed or paged copy in RAM.

\(^7\) NAND is a nonvolatile storage technology consisting of the serial alignment of the transistors. It uses tunnel injection for programming. When compared to NOR, NAND architecture enables faster contiguous data writes due to a faster erase speed. Due to reliability, NAND is best used for user data storage.
Software Spotlight
Guidelines for Choosing Bootloaders for Embedded Systems

handle bad blocks, and copy to RAM. Some systems implement this early stage of bootloading in a ROM built into the chipset.

Bootloader Misconceptions

Most “bootloaders” commonly referenced in the market are not bootloaders at all, but are instead a second-stage bootloader used to load the operating system after a first-stage bootloader has already bootstrapped the system. The most notable examples of this are desktop bootloaders such as Microsoft Windows® Boot Manager (BOOTMGR) and the common GRUB loader for Linux. The traditional role of bootstrapping the system is carried out on a desktop by the system’s first-stage bootloader. Historically, this has been BIOS and its replacement EFI. The concept of using a basic, minimal bootloader to bootstrap the system and a full-featured second-stage bootloader is now prevalent in the embedded market as well.

Types of Bootloaders

There are three categories of bootloaders, which are described in detail in the following sections:

- Pre-bootloader
- First-stage bootloader
- Second-stage bootloader

Pre-Bootloaders and First-Stage Bootloaders

The concept of using a multistage boot system in an embedded system may have originated from the need to have a small code segment to initialize the NAND controller of systems that solely use NAND for code storage. Without this code segment, the system is unable to access its nonvolatile storage. The controller initialization code is typically stored in ROM, either on or off the processor. Systems using directly-accessible memory, such as PCM, need not use a multistage bootloader since there is no need to load the image into main memory before it is executed. However, to secure all stages of the boot process to lock and protect the firmware and protect digital rights management, a small pre-bootloader in ROM may be used to check the signature of the bootloader or decrypt it, which enables the system to start in a secure state.

Regardless of the origin, some chipset manufacturers provide first-stage bootloaders that exceed the feature set of initializing the NAND controller or performing security checksums of pre-bootloaders. For example, TI uses a first-stage bootloader called X-Loader that not only bootstraps the system beyond that of the on-die ROM code, but also loads a splash screen. It can also re-flash the system from other media. However, this stripped-down bootloader at less than 64KB lacks a user interface and other features that are common to a full-featured bootloader. It is likely that the use of a first-stage and second-stage bootloader will increase over time as the number of features in second-stage bootloaders increases. However, there is little to no choice when selecting a first-stage bootloader since system manufacturers are limited to what is supported by the chipset manufacturer. Systems that do

---

not have a first-stage bootloader only have a second-stage bootloader. Single-bootloader systems are arguably only those with directly-addressable nonvolatile memory.

Second-Stage Bootloaders

It may appear that the most important task of a second-stage bootloader is to load the kernel of the intended operating system. However, the primary function of the second-stage bootloader is to set up system timings and initialize any required peripherals, such as a serial port. A second-stage bootloader is often used in a system by itself without a pre-bootloader or first-stage bootloader because its functionality exceeds that of first-stage bootloaders. Loading the kernel into RAM or jumping directly to the start of the kernel for XIP kernel implementation is considered a trivial task compared to correctly setting up the RAM timings, Flash timings, system bus configurations, and any system settings that can dramatically change the system’s performance and stability. Modern second-stage bootloaders are sometimes referred to as boot monitors because they have a command-line interface for user interaction and debugging features that exceed bootstrapping the system and loading the kernel or operating system.

The bootloader relationship in systems running NOR Flash is illustrated in Figure 1 and described in Table 1.

![Figure 1. Bootloader Relationship–Simple Case for NOR Flash](image)

Table 1. Bootloader Relationship–Simple Case for NOR Flash

<table>
<thead>
<tr>
<th>Item</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bootloader</td>
<td>Performs all bootstrapping operations, all processor and memory timings, and basic MMU setup. It also enables peripherals (such as Ethernet and serial ports) and loads the kernel after performing architecture-specific kernel requirements.</td>
</tr>
<tr>
<td>Operating System Kernel</td>
<td>Sets up memory management and configures any remaining peripherals.</td>
</tr>
</tbody>
</table>

The bootloader relationship in systems running NAND Flash is illustrated in Figure 2 and described in Table 2.

![Figure 2. Bootloader Relationship–Complex Case for NAND Flash with Security](image)

Table 2. Bootloader Relationship—Complex Case for NAND Flash with Security

<table>
<thead>
<tr>
<th>Item</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Bootloader</td>
<td>Clears the pipeline, enables the NAND controller, copies the first-stage bootloader to on-die memory, and performs a security check of the first-stage bootloader.</td>
</tr>
<tr>
<td>First-Stage Bootloader</td>
<td>Configures RAM, copies the second-stage bootloader into RAM with improved NAND management, and performs second-stage bootloader security.</td>
</tr>
<tr>
<td>Second-Stage Bootloader</td>
<td>Completes all system timing changes (such as processor, bus, and memory), performs basic MMU setup, and enables any remaining required peripherals for the architecture. It also loads the kernel after performing architecture-specific kernel requirements and performs a security check on the kernel or kernel/boot file system image.</td>
</tr>
<tr>
<td>Operating System Kernel</td>
<td>Sets up memory management, configures any remaining peripherals, and runs the system in a secure state.</td>
</tr>
</tbody>
</table>

Using Pre-, First-Stage, and Second-Stage Bootloaders in the Same System

A pre-bootloader, first-stage bootloader, and second-stage bootloader can be used in conjunction in a single system. However, depending on the complexity of the system, it may not be necessary to include all three. A general guideline for when to use each type of bootloader in your system is provided in Table 3:

Table 3. Guide for When to Use Each Bootloader Type

<table>
<thead>
<tr>
<th>Category</th>
<th>Directly Addressable (NOR, PCM)</th>
<th>Not Directly Addressable (such as NAND)</th>
<th>Signed Bootloader</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pre-Bootloader</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>First-Stage</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Second-Stage</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

Selecting an Embedded Bootloader

Architecture support should be the primary concern when selecting a bootloader. Board support files and specialty features can be generated if the architecture is supported by your bootloader. When considering the importance of architecture support and debugging tools, it is surprising that among the numerous bootloaders on the market there are only three viable bootloaders of interest: Das U-Boot, BareBox, and RedBoot. Each of these may be used as a second-stage or sole bootloader in the system. Other full-featured proprietary and open source bootloaders lack the debugging features and driver support that are key to meeting time-to-market goals. There are quick-boot bootloaders on the market that boot the system faster than these, but by configuring out unused features before compiling the bootloaders described in Table 4, the boot-time is reasonable in most cases.
Table 4. Embedded Bootloaders

<table>
<thead>
<tr>
<th>Bootloader</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Das U-Boot</td>
<td>This “universal bootloader” is arguably the most popular embedded bootloader with the largest array of supported platforms and features. This loader currently supports PPC, ARM, x86, MIPS, AVR32, Blackfin®, Motorola 68000, and other architectures[^10].</td>
</tr>
<tr>
<td>BareBox</td>
<td>Originally referred to as “Das U-Boot version 2,” this software was renamed when it no longer supported its predecessor’s legacy board files. This now independent project is typically faster than U-Boot and enables drivers to be easily ported from the Linux kernel to BareBox to reduce development time. This is recommended for chipset manufacturers with new designs and system manufacturers when the instruction set is fully supported. BareBox currently supports PPC, ARM, x86, MIPS, Blackfin, and other architectures. However, it lacks support for Motorola 68000 and AVR32, which is found in Das U-Boot[^11].</td>
</tr>
<tr>
<td>RedBoot</td>
<td>This bootloader is based on the eCOS RTOS developed by RedHat, but places an emphasis on debugging features. This was originally assumed to become the industry standard, but has been overshadowed by Das U-Boot due to the array of architectures it supports. This is an excellent bootloader for systems that need extensive debugging capabilities or when eCOS support is required for the chipset or reference platform. Before selecting this bootloader, it is advised that you verify whether Das U-Boot or BareBox has better architecture support and related board support with sufficient debugging features. RedBoot currently supports PPC, ARM, x86, MIPS, and other architectures[^12].</td>
</tr>
</tbody>
</table>

[^10]: Das U-Boot source code tree: [http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=tree;f=arch;hb=HEAD](http://git.denx.de/cgi-bin/gitweb.cgi?p=u-boot.git;a=tree;f=arch;hb=HEAD)
[^11]: BareBox source code repository: [http://git.pengutronix.de/?p=barebox.git;a=tree;f=arch;hb=HEAD](http://git.pengutronix.de/?p=barebox.git;a=tree;f=arch;hb=HEAD)
[^12]: eCOS supported hardware: [http://git.pengutronix.de/?p=barebox.git;a=tree;f=arch;hb=HEAD](http://git.pengutronix.de/?p=barebox.git;a=tree;f=arch;hb=HEAD)

Booting a Linux Kernel without a Bootloader

Using the build options, functionality can be stripped from a full-featured bootloader like Das U-Boot to decrease boot time and reduce the possibility of security loopholes. However, minimal boot time is not the design focus of bootloaders intended for command line user interaction. There are lightweight bootloaders available, such as Qi that are intended to quickly bootstrap the system and load the kernel. However, this has little advantage over abandoning the bootloader entirely. Systems running a basic or even a complex RTOS are commonly booted without the aid of a bootloader. The same sans-bootloader concept can
be applied to a Linux kernel by adding the bootstrap code directly to the kernel\textsuperscript{13}. In doing so, boot time is decreased and the system can be made more secure by removing any possible security loopholes that may have existed in the bootloader during the development phase. Since bootloaders can be stripped and optimized, the key feature to using a kernel as the bootloader is to remove the redundancy of generating and maintaining drivers for the kernel and the bootloader.

The concept of using the kernel as the bootloader is furthered by Kexec\textsuperscript{14}, which is a utility that enables the kernel to load a new kernel over itself. In essence, it loads the new kernel and performs a high-speed warm reboot without the aid of a bootloader. This is ideal for systems that require a recovery mode, system updates, or development systems. A typical system may first boot into a basic recovery state. Then, after determining that a system upgrade or recovery is not needed, the system will boot into a more full-featured kernel and system image using Kexec. A small debug kernel and file system may load in place of a debugging bootloader like RedBoot, which has greater debugging capabilities, with the aid of tools like Busybox. This tool provides a small, feature-rich set of utilities for embedded systems.

Since the use of this technique is not in the mainline Linux kernel, this is only recommended for those enabling a new chipset or developers who need the shortest possible boot time. However, those seeking a near-instant boot time should first examine the use of alternative compression algorithms, compiling options, and other boot time reduction techniques\textsuperscript{15}. In addition, for systems that do not have directly addressable memory, such as NAND-based systems, the minimal pre-bootloader that enabled the NAND controller may not be able to handle the task of loading a larger image that may contain bad blocks\textsuperscript{16}. The pre-bootloader implementation must be robust enough to load an image that is not continuous due to bad blocks and wear leveling. The kernel image must also contain the algorithm to decompress itself or be stored uncompressed. The modern Linux kernel supports images that decompress themselves.

Summary

The bootloader selection criteria are quite different for chipset developers than for system designers. Chipset developers may need to implement support for an entirely new architecture, rather than simply add drivers for new peripherals. On the other hand, system designers are far more likely to need a solution with the most support for their given chipset or similar chipset and board, given the constraints of the design cycle. With this in mind, the bootloader choice is determined by the situation and not any hard and fast rules. Chipset manufactures may want to use a minimal, custom bootloader in ROM and skip the second-stage bootloader and boot directly into the kernel to reduce the number of supported

\textsuperscript{15}Boot time reduction techniques: \url{http://elinux.org/Boot_Time} \\
software packages. However, if the new chipset is an expansion or variant of an existing family of processors, Das U-boot is recommend because of the set of tools and reference platforms. For companies making reference platforms, RedBoot is an excellent choice given that enabling the eCOS RTOS is effectively bundled with the port of the platform. For companies manufacturing a final solution, BareBox is recommended for its speed and driver portability, as long as the chipset is supported. Das U-Boot is an alternate second choice.

Figure 3. Bootloader Selection Flowchart
About the Author

Software engineer **Justin Treon** brings over seven years of experience in Flash and storage software to Micron. Specializing in embedded Linux and proof-of-concept demonstrations, Justin played a key role in the development of Micron’s Advanced XIP File System (AXFS) and Technology for Memory Optimization (TMO). He also helped develop the world’s first cell phone running Phase Change Memory (PCM).