Open BIOSes for Linux

Modern systems need not be held back by a legacy boot process

Peter Seebach

August 31, 2006

On many systems, a large portion of boot time goes into providing legacy support for MS-DOS. Various projects, including LinuxBIOS and Open Firmware, are trying to replace the proprietary BIOS systems with streamlined pieces of code able to do only what is necessary to get a Linux® kernel loaded and running. This article gives a brief overview of the field.

BEEP!

While it may seem that PC hardware beeps when it is powered on as a matter of course, in fact, there's a bit of code that makes that beep happen. That piece of code is the boot firmware. On most PCs, it's called the BIOS. (The word is an acronym for basic input/output system.) The BIOS provides the underlying hardware support that early x86 operating systems used to access disks, monitors, and just about everything else.

One of the first things the BIOS does is perform various power-up tests: identifying (and possibly testing) available memory, determining clock speeds, and so on. If the tests succeed, the machine beeps once. This process is referred to as a power-on self test, or POST. Computer geeks being what they are, the term is often used as a verb: "That machine won't even POST, so we should swap the memory."

Common diagnostics include beep codes (which vary from vendor to vendor) or diagnostic codes that can be written out to a specific raw address. Some plug-in cards allow easy access to these codes; the standard solution is that diagnostic codes are written to port 80. Some manufacturers sell plug-in cards that display, in hex, whatever byte was last written to port 80. If you do any serious debugging, you will want one of these, or possibly an even more general gizmo, like the PC Weasel, which will record the last few (256) POST codes written for your reading pleasure. (See the Related topics section below for more on the PC Weasel.) Of course, the exact meanings of these codes vary from one BIOS to another, and only some vendors document them. Luckily, the open source vendors are very good about documenting them.
What has the BIOS ever done for us?

An operating system such as MS-DOS can load additional device drivers, such as CD-ROM drivers, but needs all of the hardware drivers already loaded at startup. The standard interface provided to these drivers is handled by the BIOS, and because of this, the BIOS needs to probe for devices, identify them, and possibly initialize them for use.

Likewise, the BIOS is responsible for initializing memory. Not all operating systems require that memory be initialized, but early DOS systems often did, and even today most BIOSes probably do this for compatibility's sake. This process alone can take a long time, and many modern systems allow it to be completely or partially disabled. At the same time, the BIOS will also try to identify how much memory is available. Other boot-time activities might include initializing and enabling the processor's cache, configuring dual CPUs, building a table of information about processors, building a list of PCI devices attached to the system, and even running boot ROMs provided by those devices, which may load additional drivers.

This is a lot to do. It's so much to do, in fact, that some of the systems I own can take a full minute, or longer, to complete their POST and subsequent driver initializations. The BIOS can perform a variety of hardware scans looking for bootable devices, and on some systems the BIOS can even try to perform a network boot over Ethernet. One system I own spends about five seconds initializing network boot parameters even if network booting is disabled. Annoying!

Last but not least, the BIOS does a fair amount of initialization work. Some, but not all, of this stays useful no matter what you're going to boot. Allocating interrupt requests (IRQs) to devices may actually be a useful service, because it allows the OS to just grab a list of devices and start running, without having to program them. Many devices have configuration registers to which the BIOS can write reasonable or correct values, based on settings located in the system's writeable memory. (Generally, this memory is called CMOS, although it's not strictly required that it actually be implemented using that technology.)

What happens after the BIOS is done with all of this? It finds a chunk of code somewhere (typically on a disk) and runs it, generally loading an operating system. If the operating system is DOS, or something similar to it, all of this setup work means that you can pretty much have your command prompt instantly.

Duplicating effort

But with Linux, or BSD, or Windows®, the operating system has its own drivers. So, the operating system then skims the list of attached PCI devices, and starts loading those drivers. The work the BIOS did is mostly ignored; the operating system, once it loads a SCSI driver, will do its own scan of the SCSI bus. The BIOS is doing nothing but providing information, and much of the information it provides won't even be used. All the BIOS really had to do was load that first chunk of code (called the bootstrap loader, or bootloader) and let the machine run.

So, on a fairly heavily loaded machine, you get to wait for every device to probe twice. Often, the boot ROM for a SCSI controller takes a fairly long time to scan for devices. Worse, while the boot
ROM is running, nothing else is. In contrast, a modern operating system may well load a SCSI
driver, initiate a bus reset, then continue loading other drivers and doing other work before going
back to check for devices. In short, the operating system's scan is faster than the BIOS scan.

The exact speed difference depends on the operating system. However, we know that Linux boot-
time scans can be very, very fast. And that means that the BIOS isn't just taking up half the time
from power-on to the last kernel driver loaded; it's probably taking up a lot more. This becomes
even more pronounced when less essential kernel modules (such as sound drivers) can be loaded
fairly late in the boot process, possibly coming online after more crucial things (such as Web
servers) are loaded.

What we want is a way to get the kernel loaded without waiting for the BIOS to do a lot of the setup
that the kernel is about to do -- and do faster, better, and probably more reliably.

Put the kernel in flash

There's an obvious concern that arises if we eliminate all of the device drivers from the BIOS: if
the BIOS isn't loading all of these device drivers, how can it read the kernel? A simple solution is
to use all the space freed up by eliminating device drivers to hold a minimal kernel. All this kernel
needs to have loaded into it is the device driver for the disk on which the other, loadable, modules
are being kept. Then they can be loaded dynamically, after the kernel gets started.

The size of modern BIOS flash chips makes this alternative surprisingly viable. Many systems
have one or two megabytes of flash memory available for the BIOS. Sometimes, the actual BIOS
is really that large. Other times, it's just that the chip was cheap in bulk. A fairly stripped-down
and compressed kernel can easily fit into that kind of space, and Linux boot loaders have a lot of
experience in building tiny decompressors to extract compressed kernels.

This solution is probably no good for people doing active kernel development, but for a system
where you want the fastest possible boot, it may be a good choice. The LinuxBIOS project
(see Related topics) is working on this solution, and it is probably best targeted to servers and
embedded users.

Open Firmware

One major source of boot firmware that isn't designed for MS-DOS is Open Firmware. Primarily
used by Sun and Apple, this open standard for boot firmware is designed with less of a focus
on DOS-style systems and more interest in systems such as the original Mac OS, Mac OS X,
or Solaris, which expect to do their own driver work. Open Firmware's big advantage is that it's
"write once, run anywhere": a device with a boot ROM for Open Firmware will run just fine on any
Open Firmware system with a bus that it can plug into. The catch is that very few devices built for
x86 PCs have Open Firmware boot ROMs. Still, if you can find a device that does have such a
boot ROM, it might be a good choice; this, of course, requires that you can specify your hardware
platform and standardize on particular components.

Open Firmware is probably the most hacker-friendly BIOS-style widget out there. It doesn't have
the emphasis on speed-booting that a project like LinuxBIOS has, but it's generally a lot faster
than a traditional PC BIOS, and it is particularly friendly to users wanting ways to configure their machines.

The world would be a better place if x86 vendors started using Open Firmware by default.

**Other ways of doing a free BIOS**

Even a BIOS that simply attempts to reproduce the basic functionality of a more traditional BIOS can often make things faster and more open. For instance, the time that the BIOS delays for device probes could be adjusted, if you know that a given system has no devices that take 5 to 10 seconds for a probe.

One important thing a free BIOS can do is have a bit more flexibility in building boot loaders. For instance, the OpenBIOS project has been used in conjunction with LinuxBIOS, using the LinuxBIOS low-level code, and an OpenBIOS Forth kernel as its payload, to bootstrap a system into Open Firmware. A lot of the hardest work in x86 systems is building the tiny little block of code that loads the real boot loader; a more flexible BIOS could do that work and make the problem go away.

Note that although LinuxBIOS is aimed primarily at Linux, it’s being used to boot other things; for instance, in late 2002, the LinuxBIOS people successfully booted Windows 2000 from it. More general efforts at expanding compatibility and support are ongoing, but the idea of getting away from dependency on proprietary software to boot the system stays an important one.

There are some fairly specialized, but not really free, BIOS programs out there. For instance, Soekris Engineering boxes run a BIOS called *comBIOS*, which is much simpler and smaller than a conventional BIOS, and also boots a lot faster.

**Many eyes, no documentation**

When people wrestle with computer problems, one of the most common pieces of advice they receive is that they should update the BIOS. Why? The BIOS, in theory, isn't really in use once the system is loaded. But, in fact, the initialization work done by the BIOS can be crucial. For instance, on one embedded x86 system I used, the CardBus controller was unusable because it needed to have an IRQ programmed into it; a BIOS update fixed this. In theory, perhaps, the operating system could have developed special code to recognize this exact model of CardBus controller and program an IRQ into it, but the board vendor was in a better position to write code specific to this particular board, programming the controller correctly.

The conventional wisdom is that open source systems have a better chance of fixing bugs. However, in the case of a BIOS, it’s not quite so clear-cut. After all, the original hardware vendor may have additional information that developers of a general-purpose BIOS wouldn't have. A general-purpose, free software BIOS has to be built to run on dozens of boards; a vendor, on the other hand, can make assumptions without worrying about what the code will do on different hardware.
Nonetheless, an open source approach has many advantages. For instance, the BIOS on an old Alpha system I use has support for Symbios Logic '875 SCSI controller chips. However, it has a hard-coded list of PCI vendor/product ID pairings that it knows are supported and will only work with cards on the list; a functional card not on the list is simply ignored. Because the BIOS is closed source, and indeed somewhat obfuscated, I couldn't just patch the table; I had to buy a US $220 SCSI card from a specific vendor instead of using a physically identical off-the-shelf card for US$75.

The tipping point will probably come when a motherboard vendor decides to use an open source BIOS instead of going with one of the major commercial offerings. I don't know whether, or when, that will actually happen, but it would be very nice to see a BIOS with better support and documentation.

**Making use of all of this**

For most users, this is all still a little impractical. The danger of a misflashed BIOS chip is pretty serious: your computer won't work at all until the chip is reflashed, and since the computer can't boot to flash the chip, you'd need special hardware to do that. Some systems have two flash chips and let you use one to boot and then program the other, but not all systems do this. So, for most people, experimenting with alternative BIOS designs is probably a little risky. Still, if you find that you've got a spare, old computer you're willing to risk breaking, and it's supported by one of the free BIOS projects, it might be fun to give it a try. Hobbyists may find it appealing.

People working on server farms, or embedded systems, are a lot more likely to benefit from this technology today. If time to reset is important to you, you have a big incentive to put in the time and effort to get something like this working on the hardware you're using. Furthermore, no embedded development project is any fun if you haven't done at least one crazy and unsupported thing.

In the long run, this technology will give vendors more choices. Vendors of embedded systems have more options for boot firmware, and the proliferation of options may help get us away from a small set of proprietary BIOS designs, and into a more open and competitive market.
Related topics

- "Migrating from x86 to PowerPC, Part 2: Anatomy of the Linux boot process" (developerWorks, February 2005) adds to the discussion of BIOS issues from a Linux perspective.
- Wim's BIOS page offers an FAQ and other BIOS-related resources.
- The LinuxBIOS project is directly focused on supporting Linux.
- PC Weasel is useful whether you're running a new BIOS or an old one. It gives you serial console access to a machine that thinks it has a VGA card and a keyboard, and it provides access to POST codes.
- Technologic Systems has another variant on the home-grown BIOS program, used on their very cool embedded x86 systems.
- Soekris Engineering distributes a home-grown BIOS program, and also makes pretty toys with blinkenlights.
- With IBM trial software, available for download directly from developerWorks, build your next development project on Linux.

© Copyright IBM Corporation 2006
Trademark
(www.ibm.com/developerworks/ibm/trademarks/)