A short introduction to firmware on the x86 platform

Svante Ekholm Lindahl

January 5, 2012

Contents

1 Firmware basics

Some of the firmware, hardware and software standards are presented here. Some available tools are also described. The list is not meant to be exhaustive. Focus is placed on the protocols and interfaces that are most relevant to firmware development on the x86 architecture. It is assumed that the reader is already familiar with basic computer system structures and operation. If not, chapter 2 of Silberschatz et al. [1] is recommended.

Firmware is the initial set of instructions in a computer or piece of hardware that *bootstraps* or boots the hardware. That is, it initialises the system into a working state where it can perform its proper function. During the late sixties and throughout the seventies, firmware was simply the microcode that implemented the instruction set for a processor architecture. Davidson and Shriver [2] describe firmware in their 1978 paper as *"software in the read-only control store"*. Currently, the definition encompasses a broad range of near-hardware software configurations. For a few examples, see Rosch [3], Tanembaum [4], Catsoulis [5], Tolentino [6], Sally [7] and Zimmer et al. [8].

When power is turned on to the hardware, instructions are executed starting at a predetermined memory address (the so-called *entry vector*). The entry vector usually contains a `jmp` (jump) instruction to the first code block. The firmware program might perform a *power on self test* (POST), where the basic functions of the hardware are discovered and tested. It will explore the hardware configuration and possibly initialise resources and services (device drivers). Ultimately, the firmware may hand over control of the hardware to software located in the newly initialised secondary memory, such as an operating system. Especially for lightweight embedded systems, the firmware itself instead assumes the role of operating system or default application. In figure 1, the author contrasts the layered structure of a general-purpose computer with an embedded system. This figure moves to show that firmware engineering is not the same as embedded software engineering, where the user writes applications for an embedded system, possibly with a *real-time operating system* (RTOS) underneath. Firmware engineering concerns development of software that interfaces directly with the hardware.

The firmware is stored in a non-volatile memory chip which simplified is located close to the *central processing unit* (CPU), as shown in figure 2 [7]. While firmware used to be programmed in *read-only memory* (ROM) chips, the advent of flash memory in *small outline integrated circuits* (SOIC) and similar form factors has enabled firmware reprogramming. Communication protocol standards like inter-integrated circuit (I^2C), *serial peripheral interface* bus (SPI) and *firmware hub*

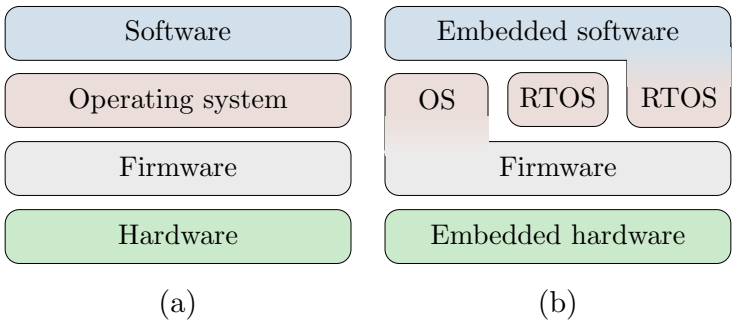


Figure 1: (a) Classical structured computer organisation. (b) In an embedded system, the layer stack might look slightly different. The firmware might assume the role of the operating system *(OS) or might be an independent layer. The embedded system might run an OS or* real-time OS *(RTOS), or the OS functionality is integrated into the embedded application.*

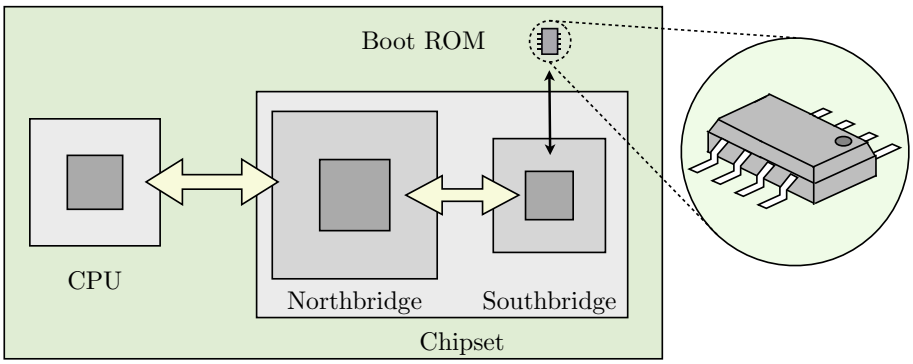


Figure 2: Simplified schematic of firmware stored in a flash memory small outline integrated circuit (SOIC) connected to the southbridge on the main circuit board. Arrows indicate interconnecting communication buses.

(FWH) are currently supported by SOIC manufacturers. The firmware can be then updated both by means of software and by means of external tools that are connected directly to the SOIC pins. Note that if the boot ROM (firmware image) is corrupt or otherwise fails to boot the device, an external tool must be used to restore the image to a working state.

1.1 Developing and debugging firmware

Firmware engineering is similar to embedded software development in two regards. First, the code deals with limited system resources and tends to be written in low-level languages such as C or assembly language. Secondly, the program is compiled on a *host* machine but executed on another *target* machine. Ganssle [9] emphasises that the code isn't magically transferred between these machines. For this reason, there is a number of hardware tools and peripherals the firmware developer must be familiar with. These tools include the external programming tool, the logic analyser, the voltmeter and the oscilloscope. They are not further explained in this text, but the reader should know of their existence.

Firmware code is inescapably hardware specific. That is, its code must be written to work with a specific configuration since the component setup and architecture instruction set differ between platforms. There is also a variety of chipsets and expansion buses, etc. Whenever new hardware components are constructed, the firmware must be ported or adapted to support the new configuration. If the new hardware is similar to the old then it is possible that the amount of work required is small. Zimmer et al. [10] note that approximately 95% of the source code of an x86 legacy BIOS can be reused for a new processor-chipset configuration. Part of the code is also universal for a specific architecture.

Rosch [3] notes that firmware traditionally is written in assembly language. With some basic chipset functionality and high memory initialised, the remaining firmware code can be written in a higher level language. A special compiler can set up a stack and enable the code to run using only the registers in the CPU [11]. Furthermore, if a cache memory is available then it could be used as a scratchpad until high memory has been initialised [8].

Davidson and Shriver [2] note that debugging firmware code is difficult due to its very nature. Before the chipset initialisation sequence is complete, the programmer has no means of gaining information of the system state. This hurdle can partly be overcome by means of virtualisation – that

is, by running the firmware on a simulated machine. Tools such as VirtualBox [12] and QEMU [13] can provide this environment. If breakpoints can be inserted into the firmware code, then the machine state, such as register values and memory contents, can be examined in detail.

Firmware often provides debugging information after the chipset has been initialised. For instance, legacy BIOS uses POST codes which are small messages that are output on *input/output* (I/O) adresses `0x80` and `0x81`. Some mainboards and add-on cards provide a translation function that outputs these messages on the serial port on I/O adress `0x3F8`, while others provide 7-segment LED displays for displaying POST codes. Some firmware types output more elaborate debug messages directly to the serial port. For example, coreboot (described in section 1.6) outputs messages in clear text while Intel's UEFI implemention outputs messages compatible with the Microsoft Windows Debugger (WinDbg) [14]. This information enables for most of the real bootstrap process to be debugged and timed, albeit in a crude way.

1.2 Classification and terminology

Jensen and Hattaway [15] split firmware or boot ROMs into two important types; multi-purpose firmware (called *BIOS*) and *boot loaders*. A BIOS provides a feature rich environment with multiple boot paths. Such an implementation is more flexible than a boot loader; it supports more configurations but has a larger memory space footprint. Reconfiguring a BIOS can be done at runtime and a plethora of options might be available. On the other hand, a boot loader is scaled-down firmware, possibly optimised for a specific (often embedded) setup. Configuration is done at compile time and the task is to boot the target system as efficiently as possible.

The term *boot loader* has many definitions. Common to all definitions is that a boot loader is known as a small piece of software that loads an operating system. Sometimes, this piece of software is located in the secondary storage and enables multiple operating systems to be installed there. It could also help load the kernel directly into memory. Such a boot loader is called a *second stage* boot loader. *Grand unified bootloader* (GRUB) and *Linux loader* (LILO) are examples of common second stage boot loaders. This text is only concerned with *first stage* boot loaders, which initialise hardware prior to passing control to an operating system or a second stage boot loader.

1.3 Common x86 interfaces

The remainder of this text will concern only the Intel x86 architecture of the target hardware. The CISC-based x86 architecture is extremely popular and is described in detail in many works such as Tanembaum [4]. It will therefore not be explained in general here. However, short introductions to some common x86 standards and interfaces are required. These include *peripheral component interconnect* (PCI) interrupt routing, the *advanced control and power interface* (ACPI) and a short mentioning of host controller interfaces.

1.3.1 Protected mode x86 interrupt handling

The x86 CPU uses hardware interrupts to signal different system events. In the 32-bit protected mode, the CPU will keep an *interrupt descriptor table* (IDT) to keep track of all the installed *interrupt handlers*. The argument supplied with the `INT` instruction is the corresponding IDT entry. The IDT thus links the IDT entry (the *interrupt vector*) to the proper device driver. For example, the instruction `INT 10h` will cause a software interrupt. It invokes the handler linked to at entry `10h` in the IDT,

the entry for the video driver. The `AX` register is used to specify the type of request and the `BX` and `CX` registers can be used to pass data parameters. There are many types of interrupts, including software interrupts, exceptions, and device interrupts caused by devices on the I/O bus.

1.3.2 PCI interrupt routing

The conventional *peripheral component interconnect* (PCI) I/O bus connect different parts of the modern x86 system into a working system. The standard does not just cover a communications protocol, is a widely used entire I/O signaling bus and describes the various physical and electrical characteristics of the PCI hardware. For a more detailed description of the PCI bus than the one given here, the textbook by Shanley and Anderson [16] is a detailed companion to the conventional PCI standard [17]. However, Baldwin [18] eminently explains the more select parts of x86 and PCI interrupt routing that are needed for firmware development. Interrupt management for *PCI express*, the successor to PCI, is mentioned on page 7.

To signal the CPU that a PCI peripheral (PCI device) needs attention, the device must utilise an *interrupt line*. The conventional PCI standard defines four interrupt lines, `INT A#` through `INT D#`. The first (and often only) function of a PCI device utilises the first line, `INT A#`. The destination for the *interrupt request* (IRQ) is the proper interrupt handler in a device driver, where an *interrupt service routine* (ISR) resolves the IRQ through some appropriate action.

All PCI device interrupt lines must be mapped to the interrupt handlers in some manner. *Interrupt controllers* exist between the devices and the CPU to facilitate this mapping. In addition, the PCI devices are physically mapped to the interrupt controllers in some configuration. This mapping can be very different from system to system. The key point here is that the firmware *must* know the corresponding logical mapping in order to correctly program the controllers [16, p. 231], or devices on the PCI bus might not function properly. There are two common interrupt controllers, called the PIC and the I/O APIC:

1. Dual `8259A` **PIC**s, or *programmable interrupt controllers*, were used in the original IBM PC with the *industrial standard architecture* (ISA) bus, a forerunner of PCI. Figure 3 (a) shows that each PIC holds eight interrupt inputs, but the output of the slave was chained to one of the inputs of the master. It is possible to chain eight slave PICs for a total of nine PICs and 64 interrupt lines [4, p. 189]. The single slave scheme became the de facto standard for interrupt routing, though, and persisted into the PCI era. As ISA did not allow IRQ sharing, most of the 15 inputs were used up by standard devices (printer, floppy, COM-ports, keyboard, etc.), leaving only 4 unused inputs for add-on cards to be used for PCI interrupt routing. Groups of PCI devices are routed through a *programmable interrupt router* (PIR)[1] to these inputs, where the members of a group share the same IRQ line. The number of inputs and outputs on PIRs varies among implementations.

2. An **I/O APIC** or *advanced* PIC moves the complexity into the controller implementation. The Intel APIC standard was developed to replace the legacy-encumbered PIC. An I/O APIC is often used in multi-processor systems and will interrupt only one of the processors. Figure 3 (b) shows that an I/O APIC is not restricted to 8 lines and it can hold an larger number (16, 24 or 32) of IRQ lines. Often the first 16 IRQ lines are programmed in the ISA manner and are then called ISA IRQs. The I/O APIC can also be used in conjunction with the legacy PICs.

[1] Note that PIR is an abbreviation for both *programmable interrupt router* and *PCI interrupt routing*.

			Pin:	Link:	Bitmap:
Bus:	Device	Type:	INT A#	0x60	0x1E39
1	0	Embedded	INT B#	0x61	0x1E39
			INT C#	0x62	0x1E39
			INT D#	0x63	0x1E39

Table 1: Example `$PIR` table entry. Here, each PCI `INTx#` is routed to a link and the bitmap shows the valid ISA IRQs for the interrupt – `0x1E39` corresponds to IRQs 3, 4, 5, 6, 9, 10, 11 and 15. The `$PIR` table contains an entry for each PCI device and slot attached to the dual PICs or *to the first 16 entries of an I/O APIC. [18]*

Type	Bus	Device	`INT` pin	Pin	APIC ID
INT	1	6	INTA#	15	0

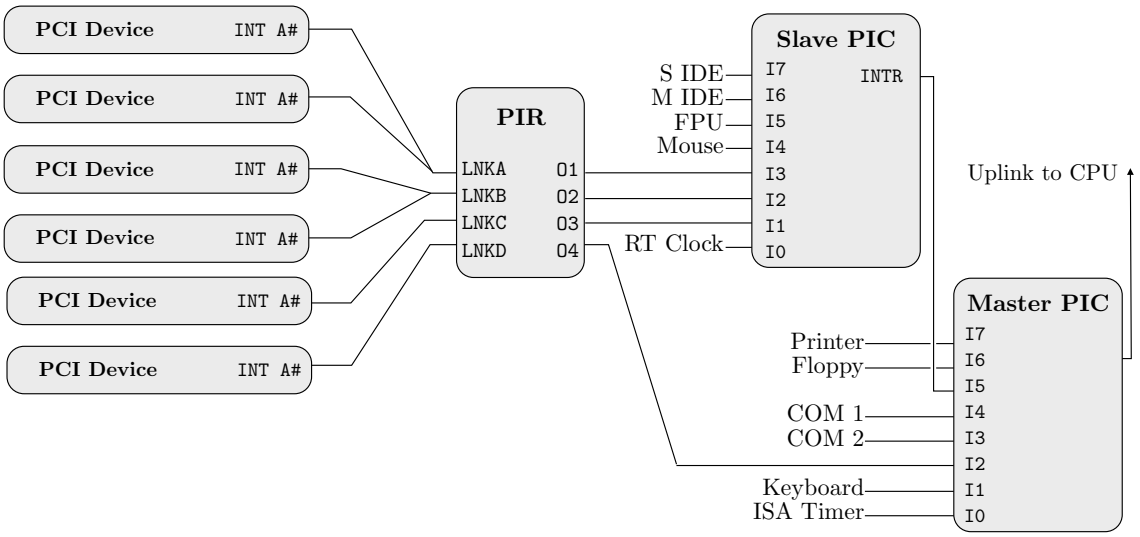
Table 2: Simplified MP table entry for the fictional PCI device 1.6, whose `INTA#` pin is connected to the sixteenth pin of the first I/O APIC.

The uplink to the CPU consists of several data lines, which the CPU not only uses to deassert interrupts after the ISR has finished, but also to update the status and registers of the interrupt controllers [4, p. 188]. In this way, the controllers can be programmed by the firmware, which can provide two tables.

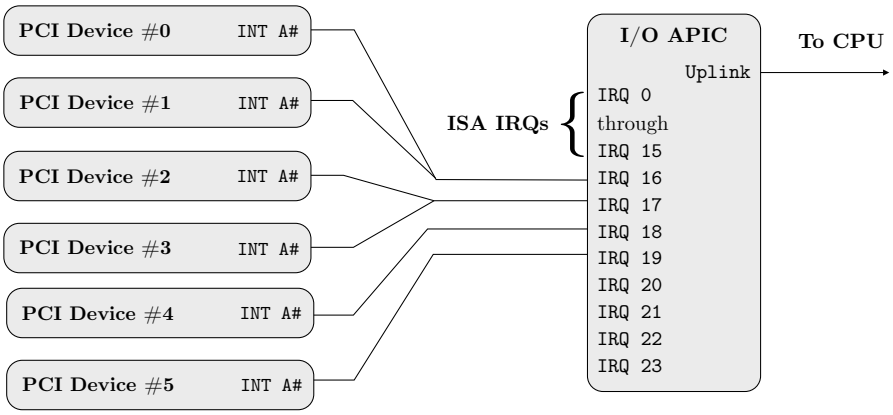
1. If dual PICs and a PIR are used, then a PCI interrupt routing (`$PIR`) table is required. The `$PIR` table is named after the ASCII signature in its header. If PICs are used in conjunction with an I/O APIC, the firmware might also need to supply a `$PIR` . The table then describes how the PCI devices are mapped to the input pins of the PIR. The `$PIR` table has one entry for each PCI device, specifying the bus and device identifiers, type (embedded or slot) and a subtable for the `INT A#-INT D#` pins. In the subtable, a link number and a bitmap of valid ISA IRQs is specified for each pin. The pin-link combinations contains the actual routing information. The bitmap could just be an opaque number, though, depending on which hardware (controllers, routers) are being used. An example table entry is shown in table 2.

2. For *multi-processor* (MP) systems or systems with an I/O APIC, the somewhat simpler MP table must be supplied. This table describes the I/O APICs in the system, among other things. Each input pin on the I/O APIC that is connected to a PCI device has an entry in the table. The interrupt entry (most importantly) specifies the bus and device numbers, the `INTx#` pin used on the device, the I/O APIC identifier, and the pin number.

Routing conventional PCI interrupts with the `$PIR` and MP tables can get quite complicated as devices often are wired in cascade. `INTA#` of the second device is normally wired to the (unused) `INTB#` of the first, and so on. This is done to avoid interrupt line sharing which, though supported, slows down a conventional PCI bus. Each new piece of hardware is also likely to be wired in new ways. The manufacturer either provides tables or schematics to the firmware vendor, or inputs tables into the firmware directly.



(a)



(b)

Figure 3: Example PCI interrupt routing. Real implementations vary between different hardware. (a) Routing with an older programmable interrupt router (PIR) and legacy `8259A` programmable interrupt controllers (PICs) [16]. (b) Routing with an advanced programmable interrupt controller (I/O APIC) [18].

PCI express interrupt handling

The newer *PCI express* (PCI-e) standard uses *message-signaled interrupts* (MSI) rather than dedicated interrupt lines. With MSI, the device will write to a register to assert an interrupt and supply a short message. If the message is the actual interrupt vector that otherwise would be stored in the IDT, the CPU can then jump directly to the interrupt handler. This means that no routing information is required for PCI-e.

1.3.3 Advanced control and power interface (ACPI)

As x86 computers grew ever more complex, the number of standards to keep track on grew larger and larger. The ACPI standard [19] unifies into a single standard a previously diverse range of standards for power management, device detection and recognition, thermal monitoring and multi-processor (MP) management, as well as PCI interrupt routing. As such, ACPI replaced both legacy Plug and Play (PnP) and the advanced power management (APM) standards.

The ACPI is an abstraction layer between the firmware and operating system that enables *operating system power management* (OSPM). It lets the operating system handle many of the things that previously needed to be handled by the firmware. The abstraction layer architecture is shown in figure 4. The *ACPI BIOS* is the part of the firmware that is compliant with the ACPI standard – note that the OS must be supplied with the ACPI tables to enable the ACPI interface. The *ACPI registers* are used for control and status purposes, as well as for events [19, p. 57]. ACPI also incorporates the `$PIR` and MP tables for convenient access by the operating system.

The ACPI tables in figure 4 enable the hardware manufacturer to describe the underlying hardware to the operating system in a coherent way across different hardware configurations. The tables are compiled from the *ACPI source language* (ASL) into *ACPI machine language* (AML). The AML is platform-independent and is parsed by an AML interpreter in the ACPI OS implementation. At

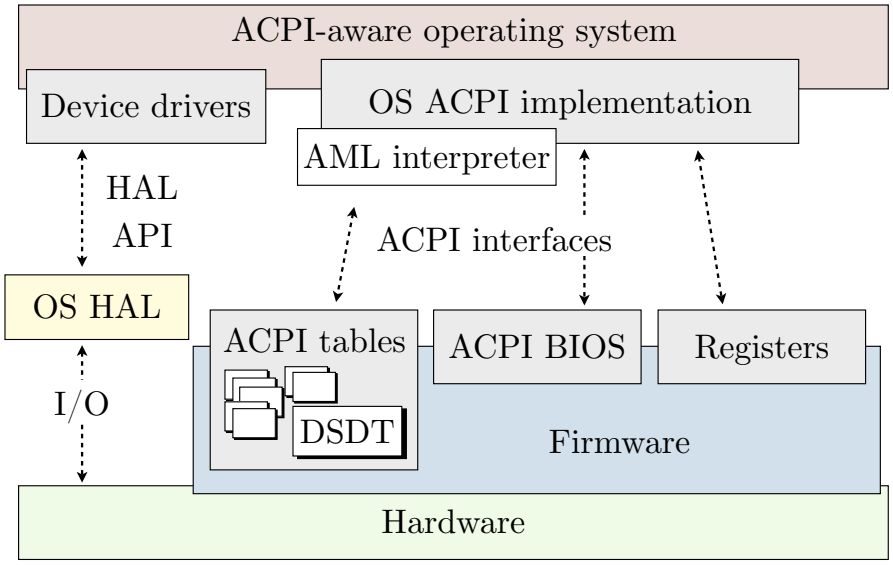


Figure 4: Simplified ACPI architecture diagram based on [19, fig. 1-1]. The interface enables transferal of system management complexity from the system firmware into the operating system. The hardware abstraction layer (HAL) and API is OS-specific.

minimum, two ACPI tables must be supplied by the OEM. The differentiated system description table (DSDT) is the most important table. Another important table is the *root system descriptor table* and its *pointer*, which together detail which tables exist. These are the tables the firmware must supply at minimum in the ACPI BIOS.

1.3.4 Host controller interfaces

A few x86-compatible host controller interfaces need to be mentioned. These controllers are often implemented as PCI devices and bridge two I/O buses. The *serial ATA* (SATA) interface for secondary storage commonly uses the *advanced host controller interface*, AHCI.

The *universal serial bus* (USB) peripheral I/O bus uses three types of controller interfaces. The *open host controller interface* (OHCI) is an open standard for USB 1.1, while the *universal host controller interface* (UHCI) is a proprietary USB 1.1 controller by Intel and often implemented in Intel chipsets. The *enhanced host controller interface* (EHCI) is a controller for the faster USB 2.0 bus.

1.4 Legacy BIOS

For the last 30 years, the 16-bit legacy BIOS has been the *de facto* standard firmware for the x86 computer architecture [20, 8]. The legacy BIOS has an interesting history of which Singh [21] provides a detailed, digestible account. The original *basic input/output system* (BIOS) [21] was 16-bit code written by IBM in 1981 for the Intel 8088 chip used in the IBM 5150. As IBM published the architecture and interfaces, other vendors were able to rewrite the BIOS and become the first *independent BIOS vendors* (IBVs). The legacy BIOS has remained the firmware base for the x86 platform ever since [8]. American Megatrends Inc. (AMI) and Phoenix Technologies Ltd. remain the two main suppliers of legacy BIOS code to OEMs [22].

The PC market dominance during the following three decades caused the word BIOS (incorrectly) to become descriptive of firmware in general. In this text, the sole word *BIOS* refers to an *unspecific, multi-purpose* firmware for the x86 or similar platform. The phrase *legacy BIOS* will refer to the 16-bit BIOS standard for the x86 platform.

Power on self test (POST) and option ROMs

The legacy BIOS default bootstrap procedure is described in Crouncher [22] and further specified in the BIOS Boot Specification [23]. When power is turned on to the system, the x86 CPU will start executing code at the entry vector `0xFFFFFFF0`, where a jump instruction moves the program counter to the POST code segment. This program initialises the hardware such as the memory controller, chipset and the I/O bus. The entire memory range might be also be checked. As executing code from the boot ROM directly is slow, the legacy BIOS is often copied to RAM once it has been initialised. The BIOS memory segment might be *shadowed*, that is mapped to the end of the address space and the address space artificially limited. The BIOS code memory range will then not be addressable by other applications.

The legacy BIOS then scans the entire memory range `0xC0000-0xEFFFF` in the boot ROM for the signature `0x55AA`, the trademark of an *option* or *expansion ROM* [16, p. 412]. These are are essentially basic *device drivers* that will initilise specific devices or components. Typically, if the system uses a PCI I/O bus, the POST program (or a PCI BIOS option ROM) will initialise the PCI bus

by scanning for PCI devices. Some PCI devices have option ROMs which are stored in and executed from the devices themselves; a typical example is a PCI video expansion card.

The option ROMs execute in 16-bit real mode and have complete control of the system during execution. Because of this, integrating the option ROM functionality into the main BIOS code keeps the BIOS developer in control. Functionality implemented in option ROMs might therefore be integrated into the main BIOS code in later versions. Some way or the other, the legacy BIOS will initialise the required data structures such as the `$PIR` , MP table, ACPI tables, etc.

OS handover

Handing over control to the operating system is done through the *master boot record* (MBR). The search is initiated by asserting the `INT 19h` interrupt. The corresponding interrupt handler in legacy BIOS will conduct a search for bootable devices by investigating the first sector (512 bytes) of all initialised secondary storage devices. The MBR table is exactly 512 bytes long and located at the beginning of sector zero of the found bootable device, the so-called *boot sector* on the *boot device*. Address `0x0` in the MBR contains a small program that searches a fixed-size data structure for operating systems in the partitions of the boot device. When the operating system is found, control is turned over to it and the BIOS boostrap process is complete.

Interrupts in the legacy BIOS

After the BIOS has performed its task, it often resides in memory to handle legacy BIOS interrupt calls. The legacy BIOS uses hardware interrupts as means of providing services to the operating system. To add support for features or hardware, the legacy BIOS source code must be changed or an option ROM must be added to the firmware. Option ROMs may provide their own interrupt handlers to handle additional function calls (or they wouldn't be very useful device drivers). A linked list of interrupt handlers is then created – this is called *chaining* an interrupt (handler). The interrupt vector in the interrupt descriptor table (IDT) entry is the memory address of noted in memory by the option ROM, which places its own interrupt vector in its place. All unrecognised interrupts are then passed to the legacy BIOS handler, creating a chain of interrupt handlers. All the chained interrupt handlers are then *hooked* at the same interrupt vector (entry) in the interrupt descriptor table (IDT). The option ROM might hook its own interrupts.

While the old *disk operating system* (DOS) relied heavily on interrupt services provided by the legacy BIOS, few modern operating systems utilise these services except during booting. These operating systems replace the legacy BIOS with their own *hardware abstraction layer* (HAL). This allows the OS to have its own direct access to the hardware. Such an OS must also implement its own driver model, but is not limited to the obsolete 16-bit real mode. If the BIOS is shadowed in RAM, then the OS can usually "de-shadow" it as the default shadow BIOS memory location is known.

Legacy BIOS video driver (VBIOS)

The most important option ROM in the legacy BIOS is the the video graphics array (VGA) option ROM or VBIOS. This is the video driver that enables the monitor during POST and is often run *before* POST so that the process can be followed by the user. Apart from the PCI option ROM signature `0x55AA`, the VBIOS also contains the phrase "IBM VGA COMPATIBLE BIOS" in standard ASCII encoding at offset `0x1e`. Bytes `0x44` through `0x47` in the file header tells the vendor ID and the

device ID of the hardware to which the binary acts as a driver. If multiple devices are found, then the legacy BIOS determines which one it should run. As the VBIOS runs, it will hook a number of `INT 10h` interrupts, which allows the operating system to change the video settings or render graphics. However, as with the other legacy BIOS drivers, the VBIOS is often later replaced with another graphics driver provided by the operating system. A modern VBIOS can initialise many different video interfaces, such as *digital visual interface* (DVI) and *low voltage differential signalling* (LVDS) – see Intel [24], for example.

Shortcomings of the legacy BIOS

Thirty years later, the 16-bit real mode legacy BIOS is still around. While it has been expanded and adapted to initialise ever more new hardware, it is facing the end of its era for several reasons.

First, the legacy BIOS consists of monolithic assembly code. Software and firmware engineering was in its cradle circa 1981, when the legacy BIOS was introduced. Davidson and Shriver [2] (1978) is an example of this. Writing long, monolithic programs in assembly language is simply put very inconvenient. The complexity of the code initialising the chipset and memory is ever increasing, making the use of assembly language impractical for the future [10, p. 50].

Secondly, the option ROMs are also restricted to 16-bit real mode. Deploying device drivers in assembly language is also inconvenient for OEMs. Giving total control of the system to the option ROM during its device or bus initialisation is bad enough, but option ROMs have also had problems regaining control at a later stage or allocating system memory safely.

Third, though assembly language has its advantages in performance over higher level languages, the 16-bit real mode severely slows down a multiprocessor 32- or 64-bit system.

Fourth, the interface to the legacy BIOS consists of real-mode interrupts and simple data structures stored at predetermined memory locations. While most operating systems provide hardware abstraction layers, this almost-static interface of the legacy BIOS prevents further vertical integration. Manageablity has also been a problem with the legacy BIOS. One would like to have better alternatives for reconfiguring or upgrading the firmware remotely, for example over a network connection.

Lastly, the legacy BIOS and its `INT 19h` OS handover call can only boot an operating system from partitions smaller than 2.2 TB [8, p. 3]. This restriction is due to the 32-bit entry for the partition

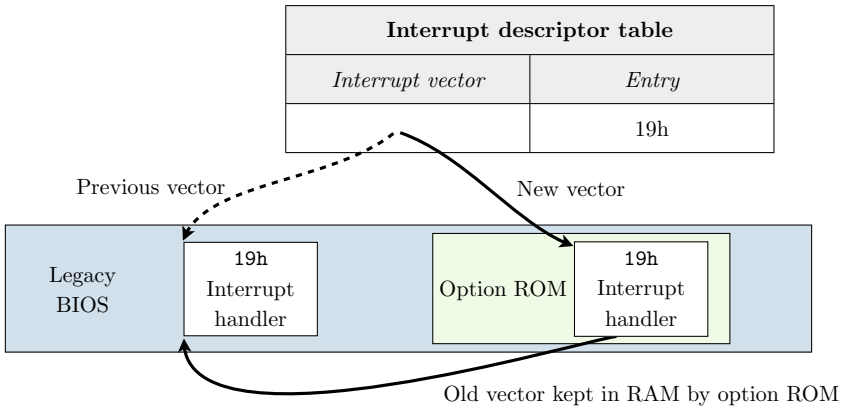


Figure 5: Expanding the legacy BIOS functionality by chaining interrupts. Here, the option ROM adds additional interrupt service routines (ISRs) to `INT 19h`, for example network device boot capabilities. If no bootable network device is found, the default ISR in the legacy BIOS is invoked.

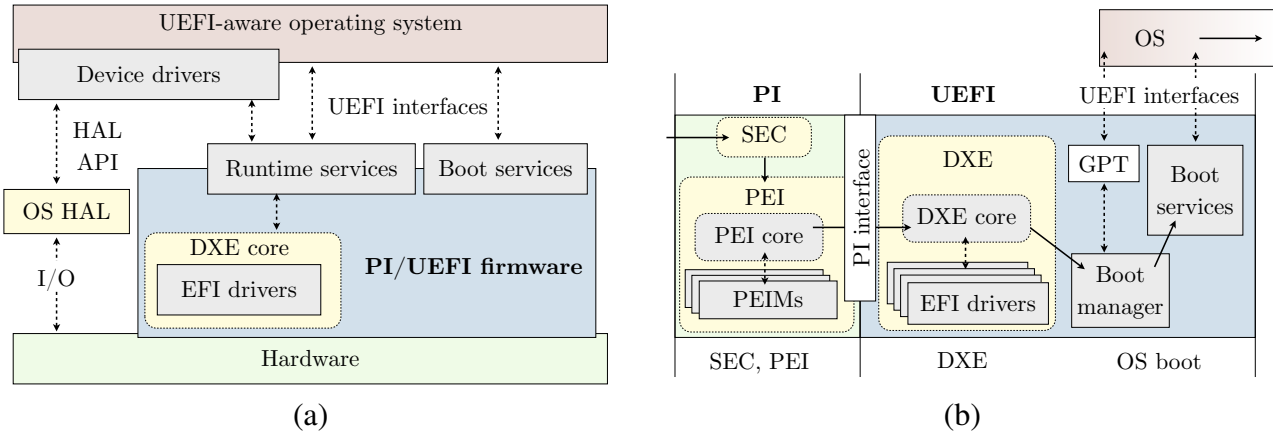


Figure 6: (a) Static UEFI architecture diagram based on [20, fig. 1]. UEFI is an interface *to which the UEFI BIOS (the implementation) conforms. The OS can interact with the UEFI BIOS runtime services or replace them with its own hardware abstraction layer (HAL). (b) Simplified dynamic UEFI architecture showing the phases during boot and order of events.*

size in the MBR. Disk drives are becoming ever larger and models breaking the 2.2 TB barrier now exist. This can be described as the firmware equivalent of running out of IPv4 addresses on the Internet.

1.5 Unified Extensible Firmware Interface (UEFI)

The UEFI standard [25] is a *firmware interface* – an abstraction layer – placed between the firmware and the operating system. It is designed to remedy most shortcomings of the legacy BIOS [20]. The interfaces defined also place requirements on how the BIOS is implemented; a UEFI compliant BIOS offers an extensible, operating system-like environment with increased functionality, security, trust and portability. The original EFI 1.0 was developed by Intel Corporation in 1999 [8, p. 6]. It became the *unified* EFI or UEFI with version 2.0. The standard has since been maintained by the UEFI forum, a consortium of several leading corporations in the hardware market. The specification itself [25], currently at version 2.3.1, is open and publicly available under a distribution restriction license. It is rather lengthy. This section provides a short introduction to the UEFI architecture and concepts. Zimmer et al. [10] is a more narrative introction to EFI and Zimmer et al. [8] an in-depth description of the standard aimed at developers of device drivers – though both are written exlusively by the inventors and promoters of UEFI.

The general architecture of an UEFI-compliant x86 system is shown in figure 6 (a). Device drivers run in a multitasking environment called the driver execution environment (DXE) [10]. In this sense, an UEFI BIOS acts much like an operating system. The figure also moves to show that the OS can interact with the UEFI BIOS runtime services and utilise some firmware drivers to the hardware, or replace them with its own hardware abstraction layer. Another way to view the UEFI firmware system is from a dynamic event view, which is shown in figure 6 (b). The first two phases are described by the platform initialisation (PI) standard, while the remaining phases fall under the UEFI standard.

Platform initialisation (PI)

The platform initialisation (PI) is a companion standard to the UEFI. Whereas UEFI defines an interface between the firmware and the operating system, PI defines an interface between the basic hardware initialisation stage and the DXE environment.

The second phase in figure 6 (b) is described first. This *pre-EFI initialisation* (PEI) phase is reponsible for bringing up system resources to the extent required by the DXE runtime environment [8, p. 267]. The chipset (northbridge, southbridge, graphics controller, memory controllers) and main memory require initialisation. Each of these components are initialised by PEI modules (PEIMs), which run under a PEIM dispatcher. The dispatcher also provides multi-tasking services to the PEIMs through the PEI *services table* such as memory allocation, inter-PEIM communications and a basic file system [8, p. 274]. Thus, a runtime environment is established very early in the boot process. This environment enables PEIMs to be written in a higher level language, such as C [8, p. 285]. Zimmer et al. [8] claim it makes firmware engineers better equipped to handle complex chipset and device functionality.

The first phase, the security phase (SEC), verifies the integrity of the various components of the following PEI phase. This phase works directly with the silicon of the CPU, similarly to a legacy BIOS (as chipset initialisation has not yet occured). The following PEI runtime environment requires a heap or stack of memory, but is also responsible for initialising the memory. Therefore, a small segment of assembly code reprograms the CPU during the SEC phase [8, p. 273]. The L1 cache memory can then be used as a scratchpad during PEI.

The DXE runtime environment

The third phase in figure 6 (b) is the *driver execution environment* phase, or DXE phase. In UEFI, the device drivers are called *EFI drivers* and *EFI applications* – or collectively as DXE drivers. DXE drivers are, like PEIMs, written in a higher level language (like C) [10, p. 53] and compiled to 32-bit x86, 64-bit x86 or 64-bit Itanium machine language. It could also be compiled into *EFI byte code* (EBC). Such programs are platform-independent and executed by an interpreter. The DXE environment is also the base for many common UEFI functions. These include but are not limited to input/output services (used for debugging) with the help of keyboards or serial ports and the UEFI network stack. With the chipset initialised, the DXE runtime environment can be (and is) more complex than the PEI environment. The DXE drivers can enable devices, buses and services [8, p. 141], including conventional PCI, PCI-e, USB, ACPI, SATA/AHCI etc. Some core services are defined as common to all UEFI systems, while other, optional drivers are added by OEMs, IBVs and hardware vendors. UEFI also allows for such programs to be run from locations *not* in the original firmware image, but also from other locations such as secondary storage or network connections. In many ways, an DXE driver is the UEFI equivalent of legacy BIOS option ROMs. The operating system-like environment also solves many of the practical problems with the legacy BIOS option ROMs. For example, the programming is done against an *application layer interface* (API) and not directly against the pure silicon of the x86 CPU. The DXE core can also be configured to dispatch only the required drivers for the boot services, leaving the rest to the operating system and reducing the time spent in firmware. Legacy option ROMs can still be run under UEFI to maintain support for legacy hardware within a UEFI BIOS, though doing so is not recommended [8, p. 312].

Internal workings of PEI/DXE handover

The internal workings of the PEI and DXE phase is shown in figure 7. Here, PI works as the interface between the bootstrap code and the following phases defined by the UEFI standard. During the PI stage, some data structures (*hand-off blocks* or HOBs) that describe the PEI modules are created. When the CPU and the on-board chips have been initialised, these contain information about the system state. The HOB list (which links to the individual HOBs) are required for passing control to the DXE phase and the initialising UEFI runtime environment.

The DXE drivers will install a number of EFI *protocols* into the UEFI *system table*. An EFI protocol is an EFI representation of a standard, such as ACPI. The representation describes the data structures and the API of the standard it implements. In fact, the DXE themselves work with a number of *architectural protocols*, the basic DXE driver API, which in turn are implemented in the DXE. These implementations in turn work directly with the hardware. Every protocol is internally identified by its *global unique identifier* or GUID [8, p. 26] and stored in the UEFI system table. Through this model, UEFI aims to be extensible and to ease the implementation of future standards.

The Tianocore open source DXE implementation

Tianocore is an open source implementation of the DXE runtime environment [8]. The Tianocore source was originally written by Intel and then put into the open source domain. The code base is provided with development tools for EFI applications and drivers, known as the EFI development kit (EDK). For example, it is used as the default UEFI runtime environment in some virtualisation software, such as QEMU [13] and Oracle's *VirtualBox* [12]. However, for use on real hardware it requires companion bootstrap code compliant with the PI standard.

Boot manager and boot services

In the UEFI environment, DXE drivers come in two flavours; boot services and runtime services [8, p. 83]. Boot services run only during the DXE phase, while runtime services can persist and coexist

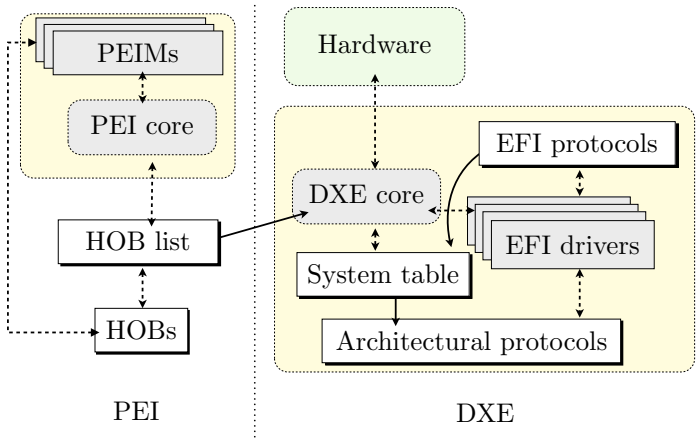


Figure 7: PEI/DXE handover diagram. The PEI dispatcher runs a number of PEI modules (PEIMs), which are described by the hand-over datablock *(HOB) structures. The HOB list is transfered into the DXE phase, where the DXE core is initialised. The latter loads the EFI drivers which install a number of EFI protocols in the UEFI system table.*

with the operating system. The operating system can then interact with these remaining services through external application layer interfaces.

Before initialising the operating system, the boot target is selected by the *boot manager*. When the DXE core considers itself done about its business, the boot manager finds and boots from a target. The boot targets and boot orders can be entered into a variable and loaded from non-volatile memory (NVRAM), similarly to how a legacy BIOS setup utility works. The UEFI boot manager works with the *GUID partition table* (GPT) as well as the master boot record (MBR) of the legacy BIOS. The GPT is part of the UEFI standard. Most importantly, the 2 TB limitation of the MBR is remedied. The GPT supports drives up to 2^{64} bytes or 16 exbibytes[2] [26]. Note, though, that the operating system must be compliant with the GPT and that a legacy BIOS also could implement GPT support.

Though EFI and UEFI firmware has existed for nearly a decade, adoption of the standard has been slow. Recently, interest in the UEFI standard has been revitalised for several reasons. First, disk drives larger than 2.2 TB began shipping during 2010. Secondly, Microsoft announced its plans for extended UEFI support in Windows 8 in September 2011 [27]. Therefore, it will likely become the dominant x86 firmware type before the end of this decade.

1.5.1 Discussion of UEFI

The UEFI standard replaces the legacy BIOS with a modern, partially open source firmware foundation. As noted in section 1.6, it will likely become the dominant x86 firmware type before the end of this decade. While UEFI is mostly aimed at chipset manufacturers and independent BIOS vendors, UEFI could also provide some flexibility for OEMs.

Control of the firmware configuration is key to utilising the UEFI interface to its maximum potential. As explained in setion 1.5, UEFI runtime services can still persist when the UEFI boot manager initiates the operating system loader. Suppose DXE drivers for specific components or devices used on OEM boards can be developed and added to the firmware. Then UEFI interfaces could facilitate runtime services as device drivers. Developing a single DXE driver might then be more cost-effective than developing the same device drivers for multiple operating systems. Note that the ability to add DXE drivers to the UEFI firmware is very important. For example, the firmware might be constructed on site using freely available tools. If a vendor UEFI BIOS is used, it must be configurable to allow drivers to load from places other than the boot ROM, for example in NVRAM. Even though the UEFI interface allows for such mechanisms, they must also be implemented so that OEMs can make use of the them.

The PEI and DXE environments are primarily designed for extensibility and flexibility. This introduces an overhead which impacts performance negatively. For an embedded system, UEFI is simply put overkill. An UEFI solution is well suited when a full-fledged BIOS, rather than a simplistic boot loader, is prefered.

[2]This will last a while: $2^{64-32} \approx 4.3 \cdot 10^9$ times the 2.2 terrabyte MBR limit.

1.6 coreboot

coreboot is an open source bootstrap code project for the x86 architecture [28]. It originates from the Advanced Computing Laboratory (ACL) at Los Alamos National Laboratory (LANL) in the United States. In 1999, Ron Minnich and other researchers there involved with high-speed data clusters were having trouble with the legacy BIOS installed on each node. In addition to the shortcomings of legacy BIOS outlined in section 1.4, Minnich found the closed source nature of the legacy BIOS problematic. Problems included the need to connect a keyboard to each of the 1024 network nodes to boot[3] or change the default firmware settings. The LANL team also required custom firmware that could be reconfigured and updated remotely. These requirements prompted the development of an open source boot loader. The resulting firmware, *LinuxBIOS*, utilised the Linux kernel as a boot loader [11]. After 8 years, the LinuxBIOS project was renamed to coreboot as the Linux kernel was no mainly longer utilised as the boot loader. Currently, coreboot is the bootstrap code segment which initialises the basic functionality of the hardware[4]. Combined with a *payload*, coreboot becomes a complete, open-source firmware solution. The payload can for example be a Linux kernel, the *SeaBIOS* open-source legacy BIOS (see page 18) or *FILO*, a boot loader which loads a Linux kernel from a file system without the help of legacy BIOS services.

Porting coreboot

Before coreboot can be used on a particular piece of hardware it must first be ported to it. coreboot currently supports over 230 motherboards. The port effort to new hardware requires detailed documentation and data sheets of the processors, chipsets and other components. CPUs and chipsets are becoming ever more complex and some even have FPGA-like elements which are programmed by bytecode in the firmware. Without the support from hardware manufacturers, this is a difficult task. It has been the author's experience that Intel Corporation is particularly secretive regarding hardware specifications. On the other hand, AMD is actively contributing to the coreboot source code and has produced a wrapper for the *AMD Generic Encapsulated Software Architecture* (AGESA) interface. AGESA is a platform initialisation API created by AMD and used to bootstrap AMD processors and chipsets. On May 5th, 2011, AMD Embedded Solutions announced that *"AMD is now committed to support coreboot for all future products on the roadmap"* [29].

Simplified, the motherboard can be viewed as a combination of different components. To add support for a new board, code must be written to initialise the CPU (which sometimes require microcode updates), northbridge and the memory controllers, southbridge and the I/O buses and possibly the *super I/O* (SIO) for a serial connection. With little or no documentation available, this can be a very time consuming process. Interrupt routing tables such as the `$PIR` and MP tables must be also supplied. Much code can be reused if the board components are already supported. The coreboot source tree contains directories for these different components. Therefore, in the simplest case, adding support for a new board is the process of combining the different pieces of the puzzle into working firmware. The file `devicetree.cb` describes the mainboard characteristics, such as which chips are on board and which I/O devices are installed. The coreboot build system parses this file and puts all the pieces together at compile time.

Both coreboot and most payloads (including SeaBIOS and FILO) are compiled with the GCC

[3] "Keyboard not found. Press F1 to continue." No joke.

[4] The bootstrap code or *core boot code* performs the same basic function as the platform initalisation (PI) implementations of the UEFI standard. However, the DXE core is not compatible with the coreboot interface.

toolchain in a Linux environment. A special cross-compiler (`xgcc`) is utilised to work around small issues with the compiler and to create the ELF executables. Debugging during the development process is done with the help of clear text messages on the COM1 serial port (I/O port `0x3F8`). If no SIO is available or the system has no serial port, it is possible to debug through a USB connection, however this requires basic initialisation of the EHCI controller.

Order of execution from entry vector to payload

As coreboot is open source, the program flow from power on to payload handoff can be followed in detail. On the Intel architecture, the coreboot bootstrap process works as follows [28, Developer manual]. The early initialisation is written in 16-bit assembly language and is followed by a mainboard initalisation stage (`mainboardinit`), written in 32-bit assembly and C. The final stage before payload handoff is called the main hardware stage, `hardwaremain`, and is written completely in C. The procedure is illustrated as a flowchart in figure 8.

1. Execution begins at the entry vector `0xFFFFFFF0` in the 32-bit x86 architecture. This address translates to the top 16 bytes of the firmware image (`0xFFFF0`, assuming an 8 Mbit image). Here, an assembly jump instruction (`reset16.inc`) is placed. Execution continues in another assembly segment, `entry16.inc`, which performs initial CPU initialisation – the translation look-aside buffers (TLBs) are turned off and a global descriptor table (GDT) is configured so that the entire memory map can be access in protected mode. The CPU then exits the 16-bit real mode and enters the 32-bit protected mode. In `entry32.inc`, the processor is further initialised by specifying the CPU segment registers.

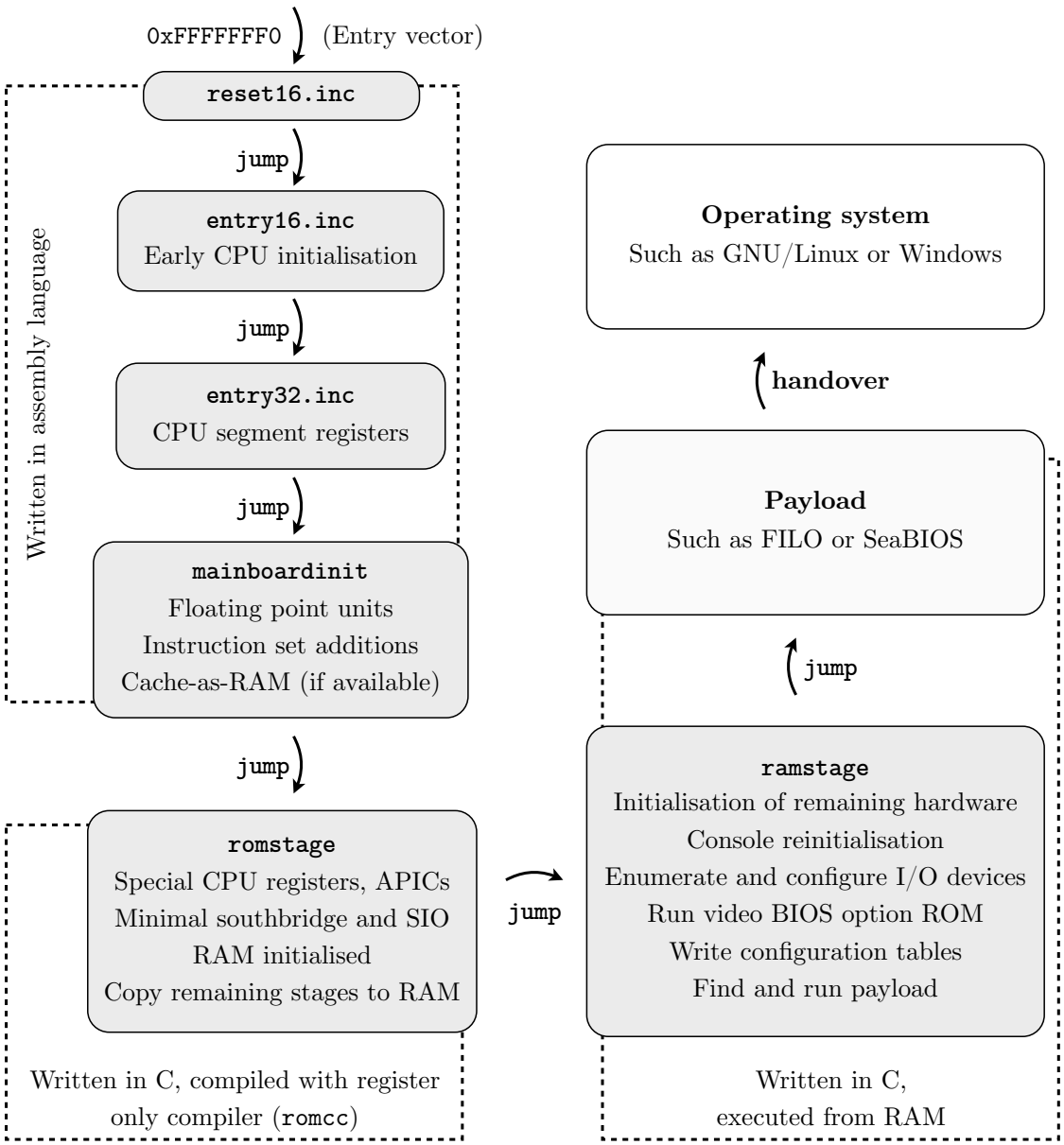
2. During the early mainboard initialisation (*mainboardinit*) stage, floating point units or instruction set additions might be initialised, depending on processor configuration. Some (well-documented) boards have support for cache-as-RAM, which is initialised at this point.

3. The late mainboard initialisation stage (the *romstage*) is written in 32-bit C. Unless the processor cache is used as RAM, it is compiled with a special register-only compiler, `romcc`. This is necessary as no RAM controller yet has been initialised. During this stage, CPU-specific and memory type registers and the internal CPU interrupt controllers (APICs) are configured. The southbridge and SIO are minimally configured and a console initiated – at this point, debug messages start to appear on the serial connection. Through the southbridge, the RAM type is identified over the system management bus (SMB) and the memory controller is initialised in the northbridge. The RAM itself is then initialised.

4. At this point, the remaining stages are copied to RAM (possibly decompressed if compression is used). The interrupt descriptor table (IDT) is initialised and the last stage, the main hardware stage (`hardwaremain`) is entered.

5. The remaining code (the *ramstage*), is 32-bit C code with a full stack in RAM. The console is reinitialised and all the devices on the I/O bus are enumerated, configured and enabled. The list of found devices is compared to the predetermined `devicetree.cb` list. Optionally, coreboot might run a VBIOS at this point to enable video output.

6. Finally, control of the system is transfered to the payload. The coreboot image (that goes into the boot ROM) uses the (rather undocumented) *coreboot file system* (CBFS). Payloads are



1

Figure 8: Flowchart of the program flow from power on to operating system handover using coreboot.

stored as ELF executables and are called (jumped to) when coreboot has initialised the basic system functionality. The payloads themselves are responsible for extracting the *CBFS tables* from memory, which contain the information regarding the system state.

The SeaBIOS payload

The SeaBIOS project is an open source implementation of basic legacy BIOS functionality, mostly written in 32-bit C by Kevin O'Connor [30]. It provides most services a modern operating system would expect from a legacy BIOS, including ACPI, SATA/AHCI (since version 1.6.2) and SMBIOS. SeaBIOS was originally developed as the default firmware for the QEMU project [13], a virtual machine environment. It has since been converted into a payload for use on real hardware in conjunction with coreboot. SeaBIOS has not been extensively tested on real hardware due to its emulator origins but development is active. SeaBIOS supports only the legacy MBR for operating system handoff, which also works with the LILO and GRUB software second stage boot loaders. Booting is supported from PATA, SATA, ATAPI CD-ROMs, USB hard drives, etc. SeaBIOS can also be configured with a splash screen (showing either JPEG or BMP images) and a boot menu. If either is used the bootstrap process will stop while the image or boot menu text is displayed.

As a payload, SeaBIOS gains control of the hardware and initialises the legacy BIOS services after coreboot passes control to it. This implies that SeaBIOS implements and hooks the corresponding interrupt service routines (ISRs) to enable an OS to boot and initialise its own hardware abstraction layer. SeaBIOS will utilise the CBFS tables if it is compiled for coreboot. SeaBIOS will also locate and extract all option ROMs from the CBFS image and execute them. For more efficient execution, option ROMs can run in "parallel" with SeaBIOS. The option ROM will execute while SeaBIOS is waiting for I/O; it is then interrupted by a 1 KHz timer IRQ set by SeaBIOS.

The Flashrom boot ROM programming utility

Flashrom is a cross-platform, open source utility for reading and writing to flash memory attached to an x86 computer system [31]. According to Borisov [32], Flashrom originated as an utility in the coreboot project and they are often used in conjunction with each other. The utility supports the common protocols *serial peripheral interface* (SPI) and *inter-integrated circuit* (I^2C), as well as the older *firmware hub* (FWH) and parallel programming (PP) protocols. Flashrom is often the software tool of choice to update the firmware on the x86 platform. This is partly because Flashrom enables the user to write anything to the non-volatile memory chip. The author has yet to find a free, commercial flash memory utility that will write arbitrary BIOS images.

1.6.1 Discussion of coreboot

coreboot is clearly an elegant and powerful open source bootstrap solution. Results in chapter **??** clearly show that coreboot and SeaBIOS can initialise a mainboard in less than a second. This is about half the time of the UEFI boot loaders developed by Insyde and described in Doran et al. [33] (though it should be noted here that different platforms were used in this comparison). The fact that coreboot is written in C also makes the source code manageable and maintainable. However, there is a serious lack of updated and approachable documentation which associates the coreboot project with a rather steep learning curve.

While coreboot supports a large number of boards, most of them tend to be of older design. It is reasonable to assume that an OEM would prefer to user newer chipsets in embedded designs. The next generation hardware usually offers better performance and power consumption efficiency. Naturally, chipset makers will continue to provide independent BIOS vendors with detailed documentation of new chipsets. The coreboot community often lacks this prerogative, so adding support for new boards can be very tedious. Moving forward, the author predicts that chipset support in coreboot will likely continue to lag behind. Developing boot loaders with coreboot becomes a more viable option if the OEM is working with long time support (LTS) boards. These are boards supported and manufactured by the board vendor for an extended period of time. That time could be utilised for open source boot loader development. In fact, the Kontron ETX-CD was an LTS board. It should be noted, though, that the author had the luxury of selecting the target hardware based only on firmware requirements.

The choice of chipset vendor is also important. AMD Embedded Solutions provides documentation freely and actively contributes with AGESA code to the coreboot project. On the other hand, Intel Embedded is extremely secretive about chipset documentation. If an OEM would pursue development of open source boot loaders then working with AMD chipsets would clearly be preferable. Working together with the hardware vendor is also an option. Unfortunately, the vendor might be opposed to the open source license of the coreboot code. There is little incentive for a company to participate if competitors sell boards on which the developed boot loader can also be applied.

coreboot only becomes a complete firmware solution when used in conjunction with a payload. SeaBIOS is a capable and efficient implementation of the legacy BIOS standard. Though not tested explicitly by the author, the FILO payload introduces an interesting way to load a Linux kernel directly into memory. Clearly there are numerous ways to load the operating system if the hardware is supported by coreboot. The difficulties lie in supplying the correct IRQ routing tables and ACPI tables – if these are incorrect it does not matter how robust the underlying firmware is.

Last, but not least, it should be noted that coreboot is free firmware – both in terms of freedom and of cost. A commercial BIOS solution is often associated with a royalty fee. For high-volume products, the potential cost savings of utilising coreboot could be rather large.

coreboot and UEFI

There currently exists no complete open source UEFI solution. Figure 9 shows how an open source, UEFI-compliant firmware alternative could be architected with the help of coreboot and Tianocore. Here, a glue code segment converts the CBFS tables and interface that describe the system state into a PI-compatible hand off blocks (HOBs). The DXE core would then assume computer control as if a closed source PI implementation had initialised the hardware. Some work has already been made in this direction, such as the works of Austin [34] and Schulz [35]. However, these projects are currently unfinished and have not been released to the public. Further research in this area is recommended.

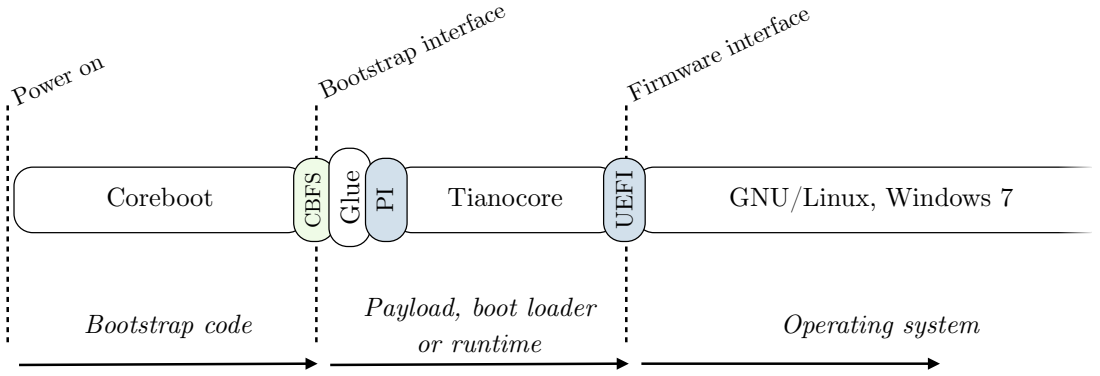


Figure 9: A theoretical open source UEFI-compliant firmware alternative: coreboot and Tianocore combination using glue code. If the CBFS tables and interface could be made into PI-compatible HOBs, then Tianocore could run after hardware initialisation by coreboot.

2 BIOS optimisation schemes

> *"Perfection is achieved, not when there is nothing left to add, but when there is nothing left to remove."*
>
> *– Antoine de Saint-Exupery*

In its broadest sense, boot time optimisation is all about streamlining the actions performed by the firmware. It is entirely possible that the firmware will perform unnecessary actions during the bootstrap process. Such actions must be identified and removed. If the firmware behaves procedurally, it might be enough to remove such actions. However, if the firmware behaves dynamically, such as the DXE runtime environment in UEFI, scheduling and dependency problems might also arise. The better the hardware configuration is known beforehand, the easier boot time optimisation becomes. Closed-box and embedded systems are therefore good targets for optimisation techniques. [36]

Recent work on BIOS optimisation schemes are presented by Doran et al. [33], Kartoz et al. [36], Rothman [37] and chapter 15 of Zimmer et al. [8]. All of these papers and chapters concern the UEFI environment, however some of the concepts should be applicable to all types of firmware:

1. BIOS boot time optimisation requires that the firmware source code is available or at least configurable. Even if source code is available, alternative firmware should always be considered. Depending on the design goals, it might or might not be a good idea to run a rather large dynamic runtime environment.

2. If a runtime environment is utilised and the drivers are ordered in a depedency tree, one must be sure to load the drivers in correct order [33]. The number of loaded drivers should also be minimised – which ones are *really* required to boot? There is, for example, no need to initialise the USB controllers unless USB keyboards during boot or USB booting are required. Video output during the boot phase might not be required; especially legacy VBIOS drivers are slow [33]. Drivers will be replaced by the OS hardware abstraction layer at a later stage [33].

3. The memory footprint of the BIOS should also be minimised. Reading the firmware from an SPI flash ROM is slow, especially if the the circuit is behind an embedded controller [33].

Compression of data could generate a performance boost if the flash ROM read operations are slow enough [36]. The overhead of decompression utilises the otherwise idle CPU time during SPI read operations. Code segments like the BIOS setup utility could be omitted to save space. As much code as possible should be executed from RAM rather than the SPI flash for faster execution speeds.

4. Code optimisations are always good. When development is complete, debugging should be turned off as the I/O operations can be extremely slow. If modules or drivers are written in a higher level language then compiler optimisations are helpful. [33]

5. The primary boot device can be set at compile time if the design requirements allow it. This eliminates the search time for a suitable boot device [36]. This idea can also be extended to memory type. The *system management bus* (SMB) on the southbridge is used to communicate with the *serial protocol detect* (SPD) module in the DRAM circuits. If the memory properties are known beforehand there is no need to initialise the SMB at runtime.

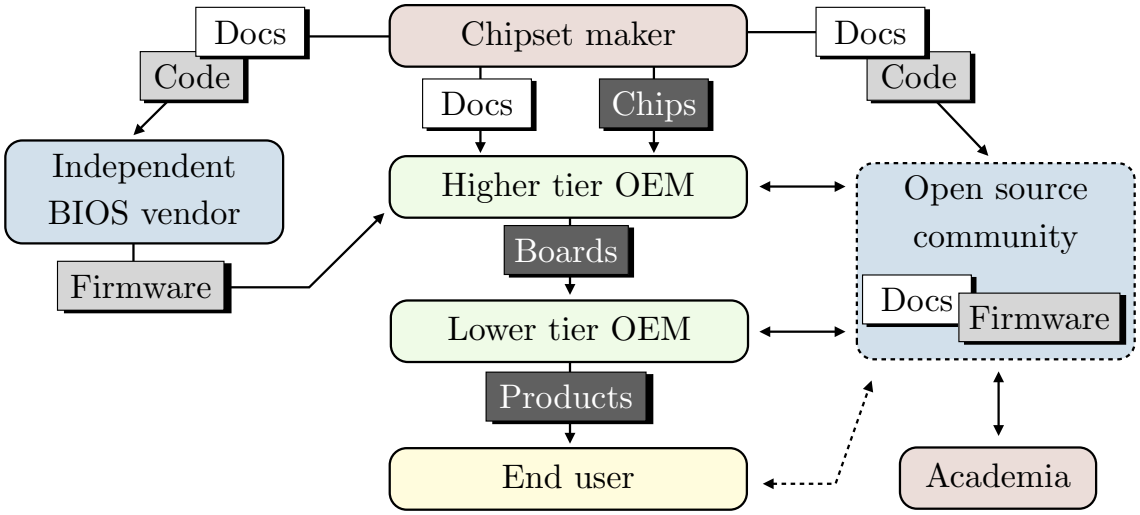


Figure 10: (Left side) Current firmware market model. In this model, the chipset maker and the BIOS vendor are controlling the firmware market. (Right side) Proposed open source-oriented firmware engineering ecosystem (coreboot, Tianocore).

3 Firmware development strategies

Firmware engineering is a complex field that scales from theoretical aspects of computer science down to the finer points of hands-on electrical engineering. Firmware development requires know-how and experience in both these fields. Despite this, firmware engineering is possible even on a lower-tier OEM (or end user) level under certain conditions. Most importantly, the *proper documentation is required*. This includes data sheets and schematics of the chipsets, boards, standards, etc. If these resources are not found within the company, then good relations to the hardware and chipset vendors are required to gain access the necessary material.

Even when the knowledge and know-how is secured, practical complications during development will likely arise. Planning ahead and allocating the proper resources helps to avoid such difficulties. ESD protection measures as well as good soldering skills are required. Tools such as logic analysers and sockets for integrated circuits can help to alleviate any practical problems with the hardware.

Note the relationships between the different parties in figure 10. The *chipset maker* sells chips and documentation to the *higher tier OEM*, which in turn sells boards to the *lower tier OEM*, which sells products to the end user. The traditional business model is depicted in the left side of the figure; the chipset maker sells documentation and source code to the *BIOS vendor*, which in turn sells the firmware or licenses the firmware source code to the higher level OEM. In contrast, an alternative ecosystem is proposed on the right side in the figure. Here, all tiers interact within the open source community. For example, this model is applicable to both the coreboot and the Tianocore projects – AMD is supplying code and documentation to coreboot while Intel is supplying both closed libraries and open source code to the Tianocore project. Note also that this model also allows academia to contribute and participate in applied firmware research.

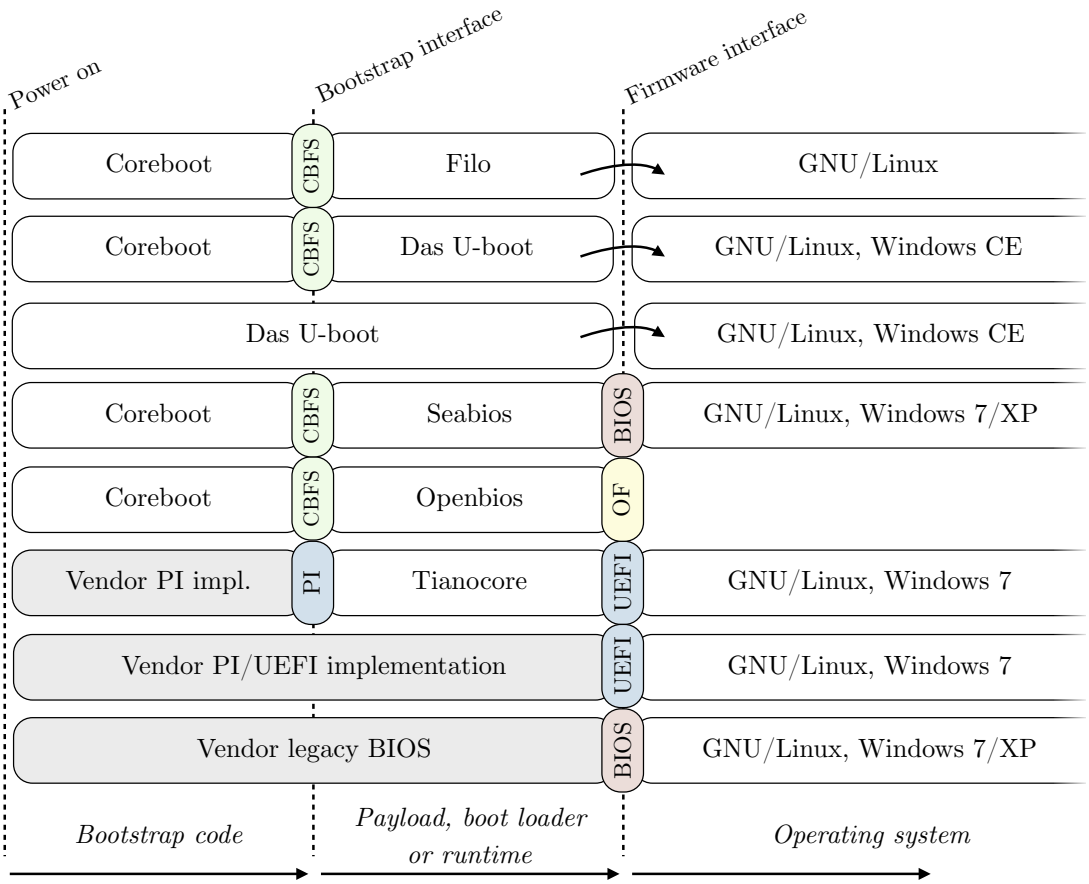


Figure 11: Comparison between bootstrap alternatives available as of 2011.

3.1 Open source bootstrap alternatives for x86

Figure 11 shows the available boostrap alternatives for the x86 platform as of 2011. White boxes in the first two stages indicate open source alternatives, while grey boxes indicate closed source. Between the different stages are clearly defined interfaces: The legacy BIOS interrupt services, the UEFI standard, the platform initalisation standard (PI), the Open Firmware standard (OF in this figure) and the CBFS (coreboot file system). If a second stage boot loader is utilised directly in the boot ROM, no services are required as the OS initialises its own hardware abstraction layer (HAL).

4 References

References

[1] A. Silberschatz, G. Gagne, and P. B. Galvin. *Operating System Concepts*. John Wiley & Sons, 1995.

[2] S. Davidson and B.D. Shriver. An Overview of Firmware Engineering. *Computer*, 11(5):21–23, 1978.

[3] W. L. Rosch. *Hardware Bible, Premier Edition*. Sams Publishing, 1997.

[4] A. S. Tanembaum. *Structured Computer Organization*. Pearson Prentice Hall, 2006.

[5] J. Catsoulis. *Designing embedded hardware*. O'Reilly Media, 2005.

[6] Matthew Tolentino. Linux in a Brave New Firmware Environment. In *Proceedings of the Linux Symposium*, volume 1, 2003.

[7] Gene Sally. *Pro Linux Embedded Systems*. Apress, 2010.

[8] V. Zimmer, M. Rothman, and S. Marisetty. *Beyond BIOS - Developing with the Unified Extensible Firmware Interface*. Intel Press, 2010.

[9] Jack G. Ganssle. *The Firmware Handbook*. Newnes, 2004.

[10] V. Zimmer, M. Rothman, and R. Hale. UEFI: From Reset Vector to Operating System. In *Hardware-Dependent Software: Principles and Practice*, 2009.

[11] D. Adhikary. Report on LinuxBIOS. Seminar, 2004.

[12] Oracle Corporation. *VirtualBox*, retrieved September 7, 2011. URL `http://www.virtualbox.org/`.

[13] Fabrice Bellard. *QEMU*, retrieved September 7, 2011. URL `http://wiki.qemu.org/Main_Page`.

[14] *Intel® Boot Loader Development Kit (Intel® BLDK) Version 2.0 – UEFI Standard Based – Getting Started Guide*. Intel Corporation, July 2011.

[15] D. Jensen and G. Hattaway. ABC's of the Intel® Boot Loader Development Kit. *EE Times*, July 2011.

[16] Tom Shanley and Don Anderson. *PCI System Architecture*. Addison Wesley, 1999.

[17] *PCI Local Bus specification*. PCI-SIG, December 1998.

[18] John Baldwin. PCI Interrupts for x86 Machines under FreeBSD. In *Proceedings of the BSDCan 2007*, May 2007.

[19] *Advanced Configuration and Power Interface Specification Rev. 2.0a*. Compaq Computer Corporation and Intel Corporation, Microsoft Corporation and Phoenix Technologies Ltd. and Toshiba Corporation, March 2002.

[20] L. Dailey Paulson. New technology beefs up BIOS. *Computer*, 37(5):22, may 2004.

[21] G. Singh. The IBM PC: The Silicon Story. *Computer*, 44(8):40 –45, August 2011.

[22] Phil Crouncher. *The BIOS Companion*. Electrocution Technical Publishers, 2004.

[23] *BIOS Boot Specification*. Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation, January 1996.

[24] *Intel® Embedded Graphics Drivers, EFI Video Driver, and Video BIOS v10.3.1 – User's Guide*. Intel Corporation, 2010.

[25] *Unified Extensible Firmware Interface Specification, Version 2.3.1*. The UEFI forum, April 2011.

[26] *FAQ: Drive Partition Limits fact sheet*. The UEFI forum, 2010.

[27] Steven Sinofsky. *Reengineering the Windows boot experience*, retrieved November 20, 2011. URL `http://blogs.msdn.com/b/b8/archive/2011/09/20/reengineering-the-windows-boot-experience.aspx`.

[28] The Coreboot project. *Coreboot*, retrieved August 30, 2011. URL `http://www.coreboot.org/`.

[29] AMD Embedded Solutions. *An Update on Coreboot*, retrieved August 30, 2011. URL `http://blogs.amd.com/work/2011/05/05/an-update-on-coreboot/`.

[30] The SeaBIOS project. *SeaBIOS*, retrieved October 22, 2011. URL `http://www.seabios.org/`.

[31] The Flashrom project. *Flashrom*, retrieved October 21, 2011. URL `http://www.flashrom.org/`.

[32] Anton Borisov. Coreboot at your service! *Linux J.*, 2009, October 2009. ISSN 1075-3583.

[33] M. Doran, K. D. Davis, and M Svancarek. UEFI Boot Time Optimization Under Windows 7. In *Intel Developer Forum 2009*, 2009.

[34] Robert Austin. *TianoCore as a payload*, retrieved November 21, 2011. URL `http://blogs.coreboot.org/blog/author/robertaustin/`.

[35] Philip Shulz. *efiboot - An UEFI payload for coreboot*, retrieved November 21, 2011. URL `http://www.phisch.org/website/efiboot/`.

[36] Mike Kartoz, Pete Dice, and Gabe Hattaway. Fastboot BIOS. White paper, Intel Corporation, September 2008.

[37] M. A. Rothman. Reducing Platform Boot Times – UEFI-based Performance Optimization. White paper, Intel Corporation, 2008.

Appendix A: Abbreviations

`$PIR`	A type of PCI interrupt routing table.
ACPI	Advanced configuration and power interface, the standard for device configuration and power management.
AGESA	AMD generic encapsulated software architecture, the bootstrap interface for AMD chipsets.
AHCI	Advanced host controller interface, a standard for configuring SATA/ACHI adapters.
AMD	Advanced Micro Devices, Inc., a hardware vendor.
AMI	American Megatrends Inc, a firmware vendor.
AML	ACPI machine language (bytecode).
API	Application program interface.
APIC	Advanced programmable interrupt controller (advanced PIC).
APM	Advanced power management, an older standard for system power management.
ARM	Advanced RISC Machine, a processor architecture.
AVR	A microcontroller family developed by Atmel.
ASL	ACPI source language (scripting language).
ATA	see PATA.
ATAPI	Modification that allows SCSI packets to be sent over IDE or SATA, used to interface with CD/DVD drives over IDE or SATA.
BIOS	Basic input/output system, a firmware implementation configurable at runtime. (The default x86 BIOS is called the *legacy BIOS* in this text.)
CBFS	Coreboot file system. CBFS works as the layout of the Coreboot boot ROM, and the interface to which payloads must comply.
CISC	Complex instruction set computing, the processor architecture paradigm of the x86 processor.
COM	Computer on module.
CPU	Central processing unit – the main processor.
DOS	Disk operating system. The original 16-bit OS for the IBM PC.
DMA	Direct memory access.
DMI	Direct media interface, an Intel bus connecting the northbridge and southbridge.
DVI	Digital visual interface, a video interface.
DRAM	Dynamic random access memory.
DSDT	Differentiated system descriptor table, part of the ACPI standard.
DXE	Driver execution environment, the runtime environment for EFI drivers.
EFI	Extensible firmware interface (superseded by UEFI).
EHCI	Enhanced host controller interface, a controller for USB 2.0.
ESD	Electrostatic discharge. Safety measures must be taken to ensure that sensitive electronics are not damaged by discharges caused by handling them.
ETX	Embedded technology extended, a COM form factor and connector standard.
FPGA	Field programmable gate array.

FSB Front-side bus, the bus traditionally connecting the CPU to the chipset.
FWH Firmware hub, an IC protocol.
GCC GNU C compiler.
GNU GNU's not UNIX.
GPT Grand partition table, the EFI and UEFI equivalent of MBR.
GUI Graphical user interface.
GUID Global unique identifier, a UEFI object key.
HOB Hand-off block, a PEI data structure also used in the PI-UEFI transition.
I/O Input / output.
I^2C Inter-integrated circuit, an IC protocol.
IASL Intel ACPI script language compiler.
IBV Independent BIOS vendor.
IC Integrated circuit.
ICH Integrated controller hub, a type of southbridge.
IDE Integrated drive electronics, another name for PATA.
IDE Integrated development environment.
IDT Interrupt descriptor table.
IRQ Interrupt request.
ISA Industrial standard architecture, an obsolete computer bus (superseded by PCI and PCI-e).
ISR Interrupt service routine.
ISP In-system programming, programming of an IC while still attached to a mainboard.
LVDS Low-voltage differential signalling.
LTS Long time support.
LZMA Lempel-Ziv-Markov chain-algorithm, a compression technique.
LPC Low pin count, an IC protocol.
MBR Master boot record.
MP Multi-processor.
MS-DOS Version of DOS supplied by Microsoft.
MSI Message signaled interrupt.
NVRAM Non-volatile RAM, a ROM where BIOS settings are stored.
OEM Original equipment manufacturer.
OS Operating system.
OSPM Operating system power management, a scheme where the OS handles power management (rather than the firmware).
OHCI Open host controller interface, an open controller standard for USB 1.1 (and others).
PATA (also IDE) Parallel ATA, an older interface for secondary storage.
PC-DOS Version of DOS supplied by IBM with the original IBM PC.
PCI Conventional peripheral component interconnect, an x86 standard I/O bus.
PCI-e PCI express, successor to conventional PCI.
PEI Pre-EFI initialisation (UEFI), a bootstrap phase during PI.
PEIM PEI module (UEFI), a device driver which runs during PI.

PI Platform initialisation, the bootstrap interface of the UEFI standard.

PIC Programmable interrupt controller.

PIR PCI interrupt routing.

PIR Programmable interrupt router.

PnP Legacy plug and play, an obsolete standard for device configuration using ISA and PCI, superseded by ACPI.

POST Power on self test.

PP Parallel programming, an older IC protocol.

PPC PowerPC, a RISC-based computer architecture.

RISC Reduced instruction set computing, a processor architecture paradigm.

ROM Read only memory.

RSTC Reset control register, a control register in contemporary Intel chipsets.

RTOS Real-time operating system.

SATA Serial ATA, a protocol for secondary storage.

SCH System controller hub. Intel term for an embedded system chipset.

sDVO Serial digital video out, a proprietary Intel video interface.

SEC Security phase (UEFI), a boot phase during PI.

SIO Super input/output, an IC providing extra I/O to the southbridge.

SMBIOS System management BIOS, an extended legacy BIOS interface for providing information about the legacy BIOS to the OS.

SMB (also SMBus) System management bus, an IC protocol.

SODIMM Small-Outline Dual In-Line Memory Module, a type of computer memory.

SOIC (also SOC) Small-Outline Integrated Circuit, an IC form factor.

SPD Serial protocol detect. Module on RAM component that contains information about the RAM.

SPARC Scalable processor architecture, a RISC-based computer architecture.

SPI Serial Peripheral Interface Bus, an IC protocol.

SST Silicon Storage Technology, an IC vendor.

TLA Three-letter abbreviation.

TLB Translation look-aside buffer.

TSOP Thin small-outline package, an IC form factor.

UEFI Unified extensible firmware interface.

UHCI Universal host controller interface, a proprietary USB controller developed by Intel.

USB Universal serial bus, a cable, connector, protocol and I/O bus standard for computer peripherals.

VBIOS (or VGA BIOS, Video BIOS) an x86 option ROM video driver.

VGA Video graphics array. An older graphics interface standard.

x86 (also Intel) PC architecture compatible with the Intel 8086 CPU.

XGA A screen resolution mode of 1024×768 pixels at 18 or 24 bits of colour depth.