FreeBSD and the IBM PC BIOS

Bruce M. Simpson
bms@FreeBSD.org

27th December 2002

1 Introduction

This document is intended as a source of technical information for individuals wishing to support FreeBSD on i386 based systems. It describes the coupling between FreeBSD and the legacy PC BIOS, and should be of interest to anybody considering implementing their own BIOS and who wishes to be able to boot FreeBSD with it. Complete listings of the BIOS routines used are provided. Interaction with the APM, ACPI, DMI, or SMBIOS subsystems is not covered, although there is brief coverage of the VESA BIOS.

For the purposes of this article, we regard the system as having completed bootstrap once the FreeBSD kernel has completed the machine-dependent part of its initialization, and before it has entered single-user mode.

2 Interaction with the BIOS during boot

On i386, FreeBSD uses a three-stage bootstrapping process. We consider the most common case; booting from a hard disk which has been recognised by the PC BIOS. The flow of execution runs as follows.

- The BIOS loads *boot0* from the Master Boot Record (MBR).
- *boot0* looks for the first FreeBSD DOS partition on the disk, which has a system ID of 0xA5; it will then load *boot1* from the BIOS Parameter Block (BPB).
- *boot1* examines the partition it has been loaded from for a BSD disk label, and the root slice within this disk label, which always has the letter "a". *boot2* is then loaded from the boot sectors at the beginning of this slice.

- *boot2* loads *loader* from UFS file system inside the root slice.
- *loader* bootstraps the FreeBSD kernel.

2.1 boot0 and boot0sio

A typical deployment of FreeBSD will install *boot0* as the bootstrap code in the MBR. It enables the user to select which BIOS partition to boot from. It is loaded from the PC BIOS "Bootstrap Loader" vector, INT 19h; upon load, the *dl* register will contain the number of the BIOS drive which is being booted from.

Once loaded, it relocates itself to segment address 0000:7C00, which is intended to be well clear of the BIOS Data Area in low memory. After relocation, it will use BIOS disk routines to load *boot1* from the selected partition's BPB into segment 0000, and invokes it by performing a near jump through the *bx* register.

boot0sio is intended for use with headless i386 systems. Rather than using the BIOS "Teletype Mode", it uses the BIOS serial port routines. It is therefore limited to the speeds which may be configured by these routines.

Table 1: BIOS Routines called by *boot0*

Int	Fn	Description
10h	0Eh	Write Character in Teletype Mode [1]
13h	02h	Read Disk Sectors
13h	03h	Write Disk Sectors
13h	42h	Extended Read Sectors
13h	43h	Extended Write Sectors
14h	00h	Initialize Serial Port [2]
14h	01h	Send Byte [2]
14h	02h	Receive Byte [2]
14h	03h	Read Status [2]
16h	00h	Read Keyboard Input [1]
16h	01h	Check Keyboard Status [1]
1Ah	00h	Get System Time

[1] Used only by boot0.
[2] Used only by boot0sio.

2.2 boot1

The BPB loader stage, *boot1*, is loaded and invoked by *boot0*. It relocates itself to segment address 0000:0700, just above the BIOS Data Area and i386 Interrupt Descriptor Table (IDT) in low memory. The A20 line is enabled via the keyboard controller to allow access to memory beyond the 1MB barrier. *boot1* then loads and relocates *boot2* to continue the boot process.

It should be noted that *boot1* is kept resident in memory to allow *boot2* to call the real-mode disk read routine, *xread*, in its segment.

Table 2: BIOS Routines called by *boot1*

Int	Fn	Description
10h	0Eh	Write Character in Teletype Mode
13h	00h	Disk Controller Reset
13h	02h	Read Disk Sectors
13h	08h	Read Disk Drive Parameters
13h	41h	Check If Extensions Present
13h	42h	Extended Read Sectors
13h	43h	Extended Write Sectors
16h	00h	Read Keyboard Input

2.3 boot2 and btxldr

/boot/boot2 is the loader found in the root slice boot blocks. It is actually a meta-binary; it contains the *boot2* loader, *btxldr*, and the *btx* framework.

btxldr is used to load the *btx* framework. It addresses video memory at 0xB8000 directly; it does not make any BIOS calls. *btx* sets up a small protected mode monitor. It is a good example of how to drive the i386. A "virtual 8086" mode monitor is used to redirect certain software interrupt vectors to the BIOS whilst remaining in protected mode. If the machine needs to be reset, INT 19h is called.

boot2 is the client of the *btx* framework. It uses the protected-mode kernel set up by *btx* to call the BIOS routines it needs. It understands enough of *ufs* to load and execute either */boot/loader* or */boot/kernel/kernel* on its own; both of which are ELF program executables.

Table 3: BIOS routines called by *boot2* components

Address	Description
40:13h	Main Memory Size
40:17h	Keyboard Shift Flags 1
40:49h	Video Mode
40:50h	Video: Cursor Position Page 0
40:72h	Warm Boot Flag

Table 4: I/O ports accessed by *boot2* components

Port	Description
0x20	8259 Master Interrupt Command Register
0x21	8259 Master Interrupt Mask Register
0xA0	8259 Slave Interrupt Command Register
0xA1	8259 Slave Interrupt Mask Register
0x3F8	UART 0 Transmit/Receive Buffer and Divisor LSB
0x3F9	UART 0 Divisor MSB
0x3FB	UART 0 Line Control
0x3FC	UART 0 Modem Control
0x3FD	UART 0 Line Status

2.4 loader

This is the final bootstrap stage for FreeBSD. *loader* consists of a portable FORTH interpreter, *ficl* [1], and various machine-dependent runtime functions. In some places in the source code, it is referred to as *boot3*. It has a command line interface, and is responsible for loading kernel modules at boot time. The kernel itself is treated as a kernel module file. It uses the /boot.config file for some settings of its own, e.g. forcing a cdrom based installation to boot without waiting for user intervention.

FreeBSD's implementation of *ficl* for i386 has functions which enable it to perform I/O directly to devices on the ISA bus. There is a wrapper for the Plug and Play BIOS (PnP) which is used less frequently on modern machines. The loader is also able to communicate with the UNDI network drivers on machines supporting the Preboot eXecution Environment (PXE) specification.

Table 5: BIOS calls intercepted by the *btxldr* virtual 8086 mode monitor

Int	Fn	Description
15h	87h	Access Extended Memory
19h	n/a	Bootstrap Loader

Table 6: BIOS calls passed through *btxldr* to the BIOS

Int	Fn	Description
10h	0Eh	Write Character in Teletype Mode
12h	88h	Base Memory Size
13h	08h	Read Disk Drive Parameters
15h	88h	Extended Memory Size
16h	00h	Read Keyboard Input
16h	01h	Check Keyboard Status

Table 7: BIOS routines called by *loader*

Int	Fn	SFn	Description
10h	02h	n/a	Set Cursor Position
10h	03h	n/a	Read Cursor Position and Type
10h	06h	n/a	Scroll Active Page Up
10h	08h	n/a	Read Character and Attribute
10h	09h	n/a	Write Character and Attribute
10h	0Eh	n/a	Write Character in Teletype Mode
12h	88h	n/a	Base Memory Size
13h	00h	n/a	Disk Controller Reset
13h	02h	n/a	Read Disk Sectors
13h	08h	n/a	Read Disk Drive Parameters
13h	42h	n/a	Extended Read Sectors
13h	4Bh	01h	Terminate Disk Emulation
15h	86h	n/a	Wait
15h	88h	n/a	Extended Memory Size
15h	E8h	01h	Get Extended Memory Info
15h	E8h	20h	Get System Memory Map
16h	00h	n/a	Read Keyboard Input
16h	01h	n/a	Check Keyboard Status
1Ah	02h	n/a	Read CMOS Time
1Ah	B1h	01h	PCI Test for PCI BIOS Present
1Ah	B1h	03h	PCI Find Device Matching Class Code
1Ah	B1h	0Ah	PCI Read Device Configuration DWORD

2.5 kernel

The kernel makes a number of BIOS calls during bootstrap. Most of these calls are made via the BIOS32 Service Directory whilst in protected mode. Other calls must be made from real mode. Few parts of the FreeBSD kernel access the BIOS directly. There are internal kernel APIs which exist to provide access to BIOS and real mode system resources; these are documented in the manual page bios(9).

Such access is generally discouraged as it is not portable to other machine architectures.

Table 8: Kernel functions providing access to real mode resources

bios16()	16-bit BIOS call from real mode. [1]
bios32()	32-bit BIOS call from protected mode.
bios32_SDlookup()	Locate a BIOS32 service.
bios_sigsearch()	Search for a service signature.
vm86_intcall()	16-bit BIOS call in V86 mode. [1]
vm86_datacall()	16-bit BIOS call in V86 mode returning data. [1]
vm86_bioscall()	16-bit BIOS call in V86 mode. [1]

The loader enters the kernel at the entry point *newboot*. Whilst bringing up the system, the kernel will reload the Global Descriptor Table (GDT). A virtual 8086 mode monitor is installed, and segment selectors are created for its use. These are later used to call the BIOS or other expansion mode ROMs from real mode whilst FreeBSD is running.

Most of the "firmware-resident" functions which the kernel calls on i386 are in the loader, not the BIOS. This design separation improves code portability between architectures. The *_bootinfo* structure is passed to the kernel by the loader; it contains most of the required information.

The kernel does make a limited number of BIOS calls during boot to discover the system's memory map. If the loader tunable *bootverbose* is set to 1, the SMAP initialization may be seen on the console.

i386 machines without ACPI support use the PCI BIOS routines to route hardware interrupts. As of this edition, ACPI is generally used for new systems.

[1] This API is undocumented.

Table 9: BIOS routines called by the kernel during boot

Int	Fn	SFn	Description
12h	88h	n/a	Base Memory Size
15h	88h	n/a	Extended Memory Size
15h	E8h	01h	Get Extended Memory Info
15h	E8h	20h	Get System Memory Map
1Ah	B1h	01h	PCI Test for PCI BIOS Present
1Ah	B1h	0Fh	PCI Route Device Interrupt

3 Interaction with the BIOS after boot

The kernel does not generally call into the BIOS after bootstrap is complete. Exceptions to this include the VESA video driver, and the AT keyboard driver. The *vm86_intcall()* function is most commonly used for this. Certain device drivers may also need to map their expansion ROMs into kernel address space, for example, hard disk controllers.

Table 10: VESA routines called by *i386/isa/vesa.c*

Int	Fn	SFn	Description
10h	00h	n/a	Set Video Mode
10h	4Fh	00h	VBE Initialize VESA BIOS
10h	4Fh	01h	VBE Get Extended Modes
10h	4Fh	02h	VBE Set Extended Mode
10h	4Fh	04h	VBE Buffer Manipulation
10h	4Fh	05h	VBE Window Manipulation
10h	4Fh	06h	VBE Scanline Manipulation
10h	4Fh	07h	VBE Linear Framebuffer Manipulation
10h	4Fh	08h	VBE DAC Manipulation
10h	4Fh	09h	VBE Palette Manipulation

Table 11: BIOS data area locations referenced by *i386/fb/vga.c*

Address	Width	Description
40:4Ah	1	Video Columns
40:4Ch	2	Video Bytes per Page
40:4Dh	2	Video Current Page Offset
40:63h	2	Video I/O Port Number Base
40:84h	1	Video Number of Rows
40:85h	2	Video Pixels per Character
40:87h	1	Video Options
40:88h	1	Video Switches
40:A8h	4	Video Parameter Control Block Pointer (DWORD)

Table 12: BIOS routines called by *dev/kbd/atkbd.c*

Int	Fn	SFn	Description
15h	C0h	n/a	Return System BIOS Configuration
16h	03h	06h	Get Typematic Rate
16h	09h	n/a	Get Typematic Capabilities

References

[1] John Sadler. Ficl - an embeddable extension language interpreter. *Dr Dobb's Journal*, January 1999.

[2] Shanley and Anderson. *PCI System Architecture*. Addison-Wesley, 4th edition, 1999. ISBN 0-201-30974-2.

[3] Michael Tischer and Bruno Jennrich. *PC Intern*. Abacus, 6th edition, 1996. ISBN 1-55755-304-1.

[4] Frank van Gilluwe. *The Undocumented PC*. Addison-Wesley, 2nd edition, 1996. ISBN 0-201-47950-8.