BIOS

From OSDev Wiki

BIOS (Basic Input/Output System) was created to offer generalized low-level services to early PC system programmers. The basic aims were: to hide (as much as possible) variations in PC models and hardware from the OS and applications, and to make OS and application development easier (because the BIOS services handled most of the hardware level interface).

These BIOS services are still used (especially during bootup), and are often named "BIOS functions". In Real Mode, they can be easily accessed through software interrupts, using Assembly language.

Contents

BIOS functions

To access a BIOS function, you generally set the AH CPU register (or AX, or EAX) to a particular value, and then do an INT opcode. The value in AH (or AX, or EAX), combined with the particular interrupt number selected requests a specific BIOS function. (Other CPU registers hold any "arguments" to the function, and often the return values from the function, also.)

It is simplest to create a listing of BIOS functions by specifying the interrupt number, and the value of AH (or AX, or EAX) that selects the function. It is also easiest to refer to particular BIOS functions this way in discussions. For example, INT 0x13, AH=0 is a BIOS function that resets hard disks or floppy disks.

Note: the INT and AH values are always listed in hexadecimal notation. Accidentally using a decimal value in an INT opcode is a very common source of errors when using BIOS functions.

To an extent, the BIOS functions are organized by interrupt number:

- INT 0x10 = Video display functions (including VESA/VBE)
- INT 0x13 = mass storage (disk, floppy) access
- INT 0x15 = memory size functions
- INT 0x16 = keyboard functions

The exhaustive list of BIOS functions is available from RBIL.

Unfortunately, the PC industry has never been good about maintaining standards. So each PC manufacturer and each BIOS manufacturer randomly made up new BIOS functions. It is also possible to "hook" any of these interrupts, and insert extra functions that mimic BIOS functions. Early PC hardware and software manufacturers did this often. So there ended up being literally thousands of BIOS functions (or mimics). The RBIL list is enormous, and is mostly filled with functions that only work when combined with some completely obsolete computer, BIOS, or piece of hardware or software.

Common functions

Unfortunately, RBIL does not clearly indicate which BIOS functions are "generic" (in some sense). That is, the ones that are always available, and that everyone uses. Partially this is because the "standard" BIOS functions grew over time, so if you go back far enough in time you can usually find a computer that does not support almost any specific BIOS function. But there is definitely a set that is commonly used in most current OSes.

- INT 0x10, AH = 1 -- set up the cursor
- INT 0x10, AH = 3 -- cursor position
- INT 0x10, AH = 0xE -- display char
- INT 0x10, AH = 0xF -- get video page and mode
- INT 0x10, AH = 0x11 -- set 8x8 font
- INT 0x10, AH = 0x12 -- detect EGA/VGA
- INT 0x10, AH = 0x13 -- display string
- INT 0x10, AH = 0x1200 -- Alternate print screen
- INT 0x10, AH = 0x1201 -- turn off cursor emulation
- INT 0x10, AX = 0x4F00 -- video memory size
- INT 0x10, AX = 0x4F01 -- VESA get mode information call
- INT 0x10, AX = 0x4F02 -- select VESA video modes
- INT 0x10, AX = 0x4F0A -- VESA 2.0 protected mode interface

- INT 0x11 -- Hardware detection

(see ATA using BIOS for more detail on these BIOS function calls)

- INT 0x13, AH = 0 -- reset floppy/hard disk
- INT 0x13, AH = 2 -- read floppy/hard disk in CHS mode
- INT 0x13, AH = 3 -- write floppy/hard disk in CHS mode
- INT 0x13, AH = 0x15 -- detect second disk
- INT 0x13, AH = 0x41 -- test existence of INT 13 extensions
- INT 0x13, AH = 0x42 -- read hard disk in LBA mode
- INT 0x13, AH = 0x43 -- write hard disk in LBA mode

(see Detecting Memory (x86) for more detail on these BIOS function calls)

- INT 0x12 -- get low memory size
- INT 0x15, EAX = 0xE820 -- get complete memory map
- INT 0x15, AX = 0xE801 -- get contiguous memory size
- INT 0x15, AX = 0xE881 -- get contiguous memory size
- INT 0x15, AH = 0x88 -- get contiguous memory size

- INT 0x15, AH = 0xC0 -- Detect MCA bus
- INT 0x15, AX = 0x0530 -- Detect APM BIOS
- INT 0x15, AH = 0x5300 -- APM detect
- INT 0x15, AX = 0x5303 -- APM connect using 32 bit
- INT 0x15, AX = 0x5304 -- APM disconnect

- INT 0x16, AH = 0 -- read keyboard scancode (blocking)
- INT 0x16, AH = 1 -- read keyboard scancode (non-blocking)
- INT 0x16, AH = 3 -- keyboard repeat rate

ASM notes

Each BIOS function (as described in RBIL) has a specific set of "result" registers. Beyond those listed registers, the BIOS functions are supposed to perfectly preserve all the other register values. Early versions of Bochs (below 2.3) had a small problem with this. The lower halves of all the 32bit extended registers (ie. EBX, ECX) were preserved properly, but the upper words of some of the registers got trashed.

The BIOS functions themselves should never crash. On any error they will:

- almost always set the carry flag (test with JC),
- sometimes return "ah = 0x86 (unsupported function)",
- sometimes return "ah = 0x80 (invalid command)"
- or (for seriously buggy BIOSes) return with nothing changed.

Try to always test these error returns, because in many circumstances the BIOS functions might appear to be returning valid (but very wrong) data -- rather than an error code.

BIOS in Protected Mode

Unfortunately, in Protected mode, almost all BIOS functions become unavailable, and trying to call them nonetheless will result in exceptions or unreliable responses (because of the different way **segment** values are handled). Some newer services however (such as SMBios, PCI, PnP, or VBE) offer an interface that is compatible with 32bit Protected Mode.

If you must use Real Mode BIOS functions after the CPU has been switched into Protected Mode, then see Virtual 8086 Mode, or perhaps exit Protected Mode, and momentarily return to Real Mode. Both methods have serious problems, and therefore any calls to the BIOS should be done before any physical device is programmed by your code:

- BIOS calls may use interrupts, which means that you need to forward IRQs or map the PIC back to its original configuration.
- BIOS calls may access devices that you have already configured - notably the PIT and PIC
- BIOS calls can enter protected mode on their own to access MMIO registers, which is beyond the limits of virtual 8086 mode.
- In real mode, you have no way of managing interrupts and your drivers may get stuck for interrupts being lost.
- In real mode, you have no control over time, performance and security guarantees.

The only device that's mostly exempt from these problems is the Video BIOS, which is not generally bundled with your motherboard and therefore can't rely on BIOS services either. Most current OSes - commercial and hobbyist alike - use a v8086 monitor or emulator to support graphics devices without a native driver so many BIOSes have been tested against such a set-up.

BIOS in Long Mode

Just like in Protected Mode, BIOS functions are unavailable in Long Mode too. Unfortunately there's no Virtual 8086 Mode to come to the rescue. It is necessary to momentarily switch to Real Mode, or to emulate a CPU and interpret opcodes by software. All necessary information for the latter approach can be found in Intel and AMD documentation.

Additional Information from the BIOS

Most of the useful information you get from the BIOS will come from calling BIOS functions. However, there is a small amount of additional information that can be acquired.

Some of the BIOS detection/state result is stored in the BIOS Data Area.

Additional information is kept in the CMOS chip.

See Also

Articles

- DMI

Threads

- ASM example code (http://forum.osdev.org/viewtopic.php?f=1&t=23125) about how to call BIOS function from Long Mode (by switching into RM and back again)

External Links

- BIOS on Wikipedia
- http://www.ctyme.com/intr/int.htm

Retrieved from "http://wiki.osdev.org/index.php?title=BIOS&oldid=16839"
Categories: X86 | BIOS | Firmware
