WIKIPEDIA

BIOS interrupt call

BIOS interrupt calls are a facility that operating systems and application programs use to invoke the facilities of the Basic Input/Output System on IBM PC compatible computers. Traditionally, BIOS calls are mainly used by DOS programs and some other software such as boot loaders (including, mostly historically, relatively simple application software that boots directly and runs without an operating system—especially game software). BIOS only runs in the real address mode (Real Mode) of the x86 CPU, so programs that call BIOS either must also run in real mode or must switch from protected mode to real mode before calling BIOS and then switch back again. For this reason, modern operating systems that use the CPU in Protected Mode generally do not use the BIOS to support system functions, although some of them use the BIOS to probe and initialize hardware resources during their early stages of booting.

In all computers, software instructions control the physical hardware (screen, disk, keyboard, etc.) from the moment the power is switched on. In a PC, the BIOS, pre-loaded in ROM on the main board, takes control immediately after the processor is reset, including during power-up or when a hardware reset button is pressed. The BIOS initializes the hardware, finds, loads and runs the boot program (usually, but not necessarily, an OS loader), and provides basic hardware control to the operating system running on the machine, which is usually an operating system but may be a directly booting single software application.

For IBM's part, they provided all the information needed to use their BIOS fully or to directly utilize the hardware and avoid BIOS completely, when programming the early IBM PC models (prior to the PS/2). From the beginning, programmers had the choice of using BIOS or not, on a per-hardware-peripheral basis. Today, the BIOS in a new PC still supports most, if not all, of the BIOS interrupt function calls defined by IBM for the IBM AT (introduced in 1984), along with many more newer ones, plus extensions to some of the originals (e.g. expanded parameter ranges). This, combined with a similar degree of hardware compatibility, means that most programs written for an IBM AT can still run correctly on a new PC today, assuming that the faster speed of execution is acceptable (which it typically is for all but games that use CPU-based timing). Despite the considerable limitations of the services accessed through the BIOS interrupts, they have proven extremely useful and durable to technological change.

Contents

Purpose of BIOS calls

BIOS interrupt calls perform hardware control or I/O functions requested by a program, return system information to the program, or do both. A key element of the purpose of BIOS calls is abstraction-the BIOS calls perform generally defined functions, and the specific details of how those functions are executed on the particular hardware of the

system are encapsulated in the BIOS and hidden from the program. So, for example, a program that wants to read from a hard disk does not need to know whether the hard disk is an ATA, SCSI, or SATA drive (or in earlier days, an ESDI drive, or an MFM or RLL drive with perhaps a Seagate ST-506 controller, perhaps one of the several Western Digital controller types, or with a different proprietary controller of another brand). The program only needs to identify the number of the drive it wishes to access and the address of the sector it needs to read or write, and the BIOS will take care of translating this general request into the specific sequence of elementary operations required to complete the task through the particular disk controller hardware that is connected to that drive. The program is freed from needing to know how to control at a low level every type of hard disk (or display adapter, or port interface, or real-time clock peripheral) that it may need to access. This both makes programming operating systems and applications easier and makes the programs smaller, reducing the duplication of program code, as the functionality that is included in the BIOS does not need to be included in every program that needs it; relatively short calls to the BIOS are included there instead. (In operating systems where the BIOS is not used, service calls provided by the operating system itself generally fulfill the same function and purpose.)

The BIOS also frees computer hardware designers (to the extent that programs are written to use the BIOS exclusively) from being constrained to maintain exact hardware compatibility with old systems when designing new systems, in order to maintain compatibility with existing software. For example, the keyboard hardware on the IBM PCjr works very differently than the keyboard hardware on earlier IBM PC models, but to programs that use the keyboard only through the BIOS, this difference is nearly invisible. (As a good example of the other side of this issue, a significant share of the PC programs in use at the time the PCjr was introduced did not use the keyboard through BIOS exclusively, so IBM also included hardware features in the PCjr to emulate the way the original IBM PC and IBM PC XT keyboard hardware works. The hardware emulation is not exact, so not all programs that try to use the keyboard hardware directly will work correctly on the PCjr, but all programs that use only the BIOS keyboard services will.)

In addition to giving access to hardware facilities, BIOS provides added facilities that are implemented in the BIOS software. For example, the BIOS maintains separate cursor positions for up to eight text display pages and provides for TTY-like output with automatic line wrap and interpretation of basic control characters such as carriage return and line feed, whereas the CGA-compatible text display hardware has only one global cursor and cannot automatically advance the cursor or interpret control characters.

Calling BIOS: BIOS software interrupts

Operating systems and other software communicates with the BIOS software, in order to control the installed hardware, via software interrupts. A software interrupt is a specific variety of the general concept of an interrupt. An interrupt is a mechanism by which the CPU can be directed to stop executing the main-line program and immediately execute a special program, called an Interrupt Service Routine (ISR), instead. Once the ISR finishes, the CPU continues with the main program. On x86 CPUs, the ISR to call when an interrupt occurs is found by looking it up in a table of their addresses (called "interrupt vectors") in memory. An interrupt is invoked by its type number, from 0 to 255; the type number is used as an index into this table; the address found in the table is the address of the ISR that will be run in response to the interrupt. A software interrupt is simply an interrupt that is triggered by a software command; therefore, software interrupts function like subroutines, with the main difference that the program that makes a software interrupt call does not need to know the address of the ISR, only its interrupt number. This has advantages for modularity, compatibility, and flexibility in system configuration.

BIOS interrupt calls can be thought of as a mechanism for passing messages between BIOS and the operating system or other BIOS client software. The messages request data or action from BIOS and return the requested data, status information, and/or the product of the requested action to the caller. The messages are broken into categories, each with its own interrupt number, and most categories contain sub-categories, called "functions" and identified by "function numbers". A BIOS client passes most information to BIOS in CPU registers, and receives most information back the same way, but data too large to fit in registers, such as tables of control parameters or disk sector data for disk transfers, is passed by allocating a buffer (i.e. some space) in memory and passing the address of the buffer in

registers. (Sometimes multiple addresses of data items in memory may be passed in a data structure in memory, with the address of that structure passed to BIOS in registers.) The interrupt number is specified as the parameter of the software interrupt instruction (in Intel assembly language, an "INT" instruction), and the function number is specified in the AH register; that is, the caller sets the AH register to the number of the desired function. In general, the BIOS services corresponding to each interrupt number operate independently of each other, but the functions within one interrupt service are handled by the same BIOS program and are not independent. (This last point is relevant to reentrancy.)

The BIOS software usually returns to the caller with an error code if not successful, or with a status code and/or requested data if successful. The data itself can be as small as one bit or as large as 65,536 bytes of whole raw disk sectors (the maximum that will fit into one real-mode memory segment). BIOS has been expanded and enhanced over the years many times by many different corporate entities, and unfortunately the result of this evolution is that not all the BIOS functions that can be called use consistent conventions for formatting and communicating data or for reporting results. Some BIOS functions report detailed status information, while others may not even report success or failure but just return silently, leaving the caller to assume success (or to test the outcome some other way). Sometimes it can also be difficult to determine whether or not a certain BIOS function call is supported by the BIOS on a certain computer, or what the limits of a call's parameters are on that computer.

Because BIOS interrupt calls use CPU register-based parameter passing, the calls are oriented to being made from assembly language and cannot be directly made from most high-level languages (HLLs). However, a high level language may provide a library of wrapper routines which translate parameters from the form (usually stack-based) used by the high-level language to the register-based form required by BIOS, then back to the HLL calling convention after the BIOS returns. In some variants of C, BIOS calls can be made using inline assembly language within a C module. (Support for inline assembly language is not part of the ANSI C standard but is a language extension; therefore, C modules that use inline assembly language are less portable than pure ANSI standard C modules.)

Invoking an interrupt

Invoking an interrupt can be done using the INT x86 assembly language instruction. For example, to print a character to the screen using BIOS interrupt 0x10, the following x86 assembly language instructions could be executed:

```
mov ah, 0x0e    ; function number = 0Eh : Display Character
mov al, '!'     ; AL = code of character to display
int 0x10        ; call INT 10h, BIOS video service
```

Interrupt table

A list of common BIOS interrupt classes can be found below. Note that some BIOSes (particularly old ones) do not implement all of these interrupt classes.

The BIOS also uses some interrupts to relay hardware event interrupts to programs which choose to receive them or to route messages for its own use. The table below includes only those BIOS interrupts which are intended to be called by programs (using the "INT" assembly-language software interrupt instruction) to request services or information.

<table>
<tr><th>Interrupt vector</th><th>Description</th></tr>
<tr><td>05h</td><td>Executed when Shift-Print screen is pressed, as well as when the BOUND instruction detects a bound failure.</td></tr>
<tr><td>10h</td><td>Video Services
<table>
<tr><th>AH</th><th>Description</th></tr>
<tr><td>00h</td><td>Set Video Mode</td></tr>
<tr><td>01h</td><td>Set Cursor Shape</td></tr>
<tr><td>02h</td><td>Set Cursor Position</td></tr>
<tr><td>03h</td><td>Get Cursor Position And Shape</td></tr>
<tr><td>04h</td><td>Get Light Pen Position</td></tr>
<tr><td>05h</td><td>Set Display Page</td></tr>
<tr><td>06h</td><td>Clear/Scroll Screen Up</td></tr>
<tr><td>07h</td><td>Clear/Scroll Screen Down</td></tr>
<tr><td>08h</td><td>Read Character and Attribute at Cursor</td></tr>
<tr><td>09h</td><td>Write Character and Attribute at Cursor</td></tr>
<tr><td>0Ah</td><td>Write Character at Cursor</td></tr>
<tr><td>0Bh</td><td>Set Border Color</td></tr>
<tr><td>0Ch</td><td>Write Graphics Pixel</td></tr>
<tr><td>0Dh</td><td>Read Graphics Pixel</td></tr>
<tr><td>0Eh</td><td>Write Character in TTY Mode</td></tr>
<tr><td>0Fh</td><td>Get Video Mode</td></tr>
<tr><td>10h</td><td>Set Palette Registers (EGA, VGA, SVGA)</td></tr>
<tr><td>11h</td><td>Character Generator (EGA, VGA, SVGA)</td></tr>
<tr><td>12h</td><td>Alternate Select Functions (EGA, VGA, SVGA)</td></tr>
<tr><td>13h</td><td>Write String</td></tr>
<tr><td>1Ah</td><td>Get or Set Display Combination Code (VGA, SVGA)</td></tr>
<tr><td>1Bh</td><td>Get Functionality Information (VGA, SVGA)</td></tr>
<tr><td>1Ch</td><td>Save or Restore Video State (VGA, SVGA)</td></tr>
<tr><td>4Fh</td><td>VESA BIOS Extension Functions (SVGA)</td></tr>
</table></td></tr>
<tr><td>11h</td><td>Returns equipment list</td></tr>
<tr><td>12h</td><td>Return conventional memory size</td></tr>
<tr><td>13h</td><td>Low Level Disk Services</td></tr>
</table>

AH	Description
00h	Reset Disk Drives
01h	Check Drive Status
02h	Read Sectors
03h	Write Sectors
04h	Verify Sectors
05h	Format Track
08h	Get Drive Parameters
09h	Init Fixed Drive Parameters
0Ch	Seek To Specified Track
0Dh	Reset Fixed Disk Controller
15h	Get Drive Type
16h	Get Floppy Drive Media Change Status
17h	Set Disk Type
18h	Set Floppy Drive Media Type
41h	Extended Disk Drive (EDD) Installation Check
42h	Extended Read Sectors
43h	Extended Write Sectors
44h	Extended Verify Sectors
45h	Lock/Unlock Drive
46h	Eject Media
47h	Extended Seek
48h	Extended Get Drive Parameters
49h	Extended Get Media Change Status
4Eh	Extended Set Hardware Configuration

14h

Serial port services

AH	Description
00h	Serial Port Initialization
01h	Transmit Character
02h	Receive Character
03h	Status

15h

Miscellaneous system services

AH	AL	Description
00h		Turn on cassette drive motor (IBM PC/PCjr only)
01h		Turn off cassette drive motor (IBM PC/PCjr only)
02h		Read data blocks from cassette (IBM PC/PCjr only)
03h		Write data blocks to cassette (IBM PC/PCjr only)
4Fh		Keyboard Intercept
83h		Event Wait
84h		Read Joystick (BIOSes from 1986 onward)
85h		Sysreq Key Callout
86h		Wait
87h		Move Block
88h		Get Extended Memory Size
89h		Switch to Protected Mode
C0h		Get System Parameters
C1h		Get Extended BIOS Data Area Segment
C2h		Pointing Device Functions
C3h		Watchdog Timer Functions - PS/2 systems only
C4h		Programmable Option Select - MCA bus PS/2 systems only
D8h		EISA System Functions - EISA bus systems only
E8h	01h	Get Extended Memory Size (Newer function, since 1994). Gives results for memory size above 64 Mb.
E8h	20h	Query System Address Map. The information returned from E820 supersedes what is returned from the older AX=E801h and AH=88h interfaces.

16h

Keyboard services

AH	Description
00h	Read Character
01h	Read Input Status
02h	Read Keyboard Shift Status
05h	Store Keystroke in Keyboard Buffer
10h	Read Character Extended
11h	Read Input Status Extended
12h	Read Keyboard Shift Status Extended

17h

Printer services

AH	Description
00h	Print Character to Printer
01h	Initialize Printer
02h	Check Printer Status

18h

Execute Cassette BASIC: On IBM machines up to the early PS/2 line, this interrupt would start the

<table>
<tr><td></td><td>ROM Cassette BASIC. Clones did not have this feature and different machines/BIOSes would perform a variety of different actions if INT 18h was executed, most commonly an error message stating that no bootable disk was present. Modern machines would attempt to boot from a network through this interrupt.</td></tr>
<tr><td>19h</td><td>After POST this interrupt is used by the BIOS to load the operating system. A program can call this interrupt to reboot the computer (but must ensure that hardware interrupts or DMA operations will not cause the system to hang or crash during either the reinitialization of the system by BIOS or the boot process).</td></tr>
<tr><td>1Ah</td><td>Real Time Clock Services
<table>
<tr><th>AH</th><th>Description</th></tr>
<tr><td>00h</td><td>Read RTC</td></tr>
<tr><td>01h</td><td>Set RTC</td></tr>
<tr><td>02h</td><td>Read RTC Time</td></tr>
<tr><td>03h</td><td>Set RTC Time</td></tr>
<tr><td>04h</td><td>Read RTC Date</td></tr>
<tr><td>05h</td><td>Set RTC Date</td></tr>
<tr><td>06h</td><td>Set RTC Alarm</td></tr>
<tr><td>07h</td><td>Reset RTC Alarm</td></tr>
</table></td></tr>
<tr><td>1Ah</td><td>PCI Services - implemented by BIOSes supporting PCI 2.0 or later
<table>
<tr><th>AX</th><th>Description</th></tr>
<tr><td>B101h</td><td>PCI Installation Check</td></tr>
<tr><td>B102h</td><td>Find PCI Device</td></tr>
<tr><td>B103h</td><td>Find PCI Class Code</td></tr>
<tr><td>B106h</td><td>PCI Bus-Specific Operations</td></tr>
<tr><td>B108h</td><td>Read Configuration Byte</td></tr>
<tr><td>B109h</td><td>Read Configuration Word</td></tr>
<tr><td>B10Ah</td><td>Read Configuration Dword</td></tr>
<tr><td>B10Bh</td><td>Write Configuration Byte</td></tr>
<tr><td>B10Ch</td><td>Write Configuration Word</td></tr>
<tr><td>B10Dh</td><td>Write Configuration Dword</td></tr>
<tr><td>B10Eh</td><td>Get IRQ Routine Information</td></tr>
<tr><td>B10Fh</td><td>Set PCI IRQ</td></tr>
</table></td></tr>
<tr><td>1Bh</td><td>Ctrl-Break handler - called by <code>INT 09</code> when Ctrl-Break has been pressed</td></tr>
<tr><td>1Ch</td><td>Timer tick handler - called by <code>INT 08</code></td></tr>
<tr><td>1Dh</td><td>Not to be called; simply a pointer to the VPT (Video Parameter Table), which contains data on video modes</td></tr>
<tr><td>1Eh</td><td>Not to be called; simply a pointer to the DPT (Diskette Parameter Table), containing a variety of information concerning the diskette drives</td></tr>
<tr><td>1Fh</td><td>Not to be called; simply a pointer to the VGCT (Video Graphics Character Table), which contains the data for ASCII characters <code>80h</code> to <code>FFh</code></td></tr>
<tr><td>41h</td><td>Address pointer: FDPT = Fixed Disk Parameter Table (1st hard drive)</td></tr>
<tr><td>46h</td><td>Address pointer: FDPT = Fixed Disk Parameter Table (2nd hard drive)</td></tr>
</table>

4Ah	Called by RTC for alarm

INT 18h: execute BASIC

`INT 18h` traditionally jumped to an implementation of Cassette BASIC (provided by Microsoft) stored in Option ROMs. This call would typically be invoked if the BIOS was unable to identify any bootable disk volumes on startup.

At the time the original IBM PC (IBM machine type 5150) was released in 1981, the BASIC in ROM was a key feature. Contemporary popular personal computers such as the Commodore 64 and the Apple II line also had Microsoft Cassette BASIC in ROM (though Commodore renamed their licensed version Commodore BASIC), so in a substantial portion of its intended market, the IBM PC needed BASIC to compete. As on those other systems, the IBM PC's ROM BASIC served as a primitive diskless operating system, allowing the user to load, save, and run programs, as well as to write and refine them. (The original IBM PC was also the only PC model from IBM that, like its aforementioned two competitors, included cassette interface hardware. A base model IBM PC had only 16 KiB of RAM and no disk drives [*of any kind*], so the cassette interface and BASIC in ROM were essential to make the base model usable. Of the five 8 KiB ROM chips in an original IBM PC, totaling 40 KiB, four contain BASIC and only one contains the BIOS; the ROM BASIC accounts for over half of the total system memory [4/7ths, to be precise].)

As time went on and BASIC was no longer shipped on all PCs, this interrupt would simply display an error message indicating that no bootable volume was found (famously, "No ROM BASIC", or more explanatory messages in later BIOS versions); in other BIOS versions it would prompt the user to insert a bootable volume and press a key, and then after the user pressed a key it would loop back to the bootstrap loader (INT 19h) to try booting again.

Digital's Rainbow 100B used `INT 18h` to call its BIOS, which was incompatible with the IBM BIOS. Turbo Pascal, Turbo C and Turbo C++ repurposed INT 18 for memory allocation and paging. Other programs also reused this vector for their own purposes.

BIOS hooks

DOS

On DOS systems, IO.SYS or IBMBIO.COM hooks INT 13 for floppy disc change detection, tracking formatting calls, correcting DMA boundary errors, and working around problems in IBM's ROM BIOS "01/10/84" with model code 0xFC before the first call.

Bypassing BIOS

Many modern operating systems (such as Linux and newer versions of Windows) bypass the built-in BIOS interrupt communication system altogether, preferring to use their own software to control the attached hardware directly. The original reason for this was primarily that these operating systems run the processor in protected mode, whereas calling BIOS requires switching to real mode and back again, and switching to real mode is slow. However, there are also serious security reasons not to switch to real mode, and the BIOS code has limitations both in functionality and speed that motivate operating system designers to find a replacement for it. In fact, the speed limitations of the BIOS made it common even in the DOS era for programs to circumvent it in order to avoid its performance limitations, especially for video graphics display and fast serial communication. The problems with BIOS functionality include limitations in the range of functions defined, inconsistency in the subsets of those functions supported on different computers, and variations in the quality of BIOSes (i.e. some BIOSes are complete and reliable, others are abridged and buggy). By taking matters into their own hands and avoiding reliance on BIOS, operating system developers can eliminate some of the risks and complications they face in writing and supporting system software. On the other hand, by doing so those developers become responsible for providing "bare-metal" driver software for every different system

or peripheral device they intend for their operating system to work with (or for inducing the hardware producers to provide those drivers). Thus it should be apparent that compact operating systems developed on small budgets would tend to use BIOS heavily, while large operating systems built by huge groups of software engineers with large budgets would more often opt to write their own drivers instead of using BIOS—that is, even without considering the compatibility problems of BIOS and protected mode.

See also

- DOS interrupt call
- Interrupt descriptor table
- Input/Output Base Address
- Ralf Brown's Interrupt List

References

- The x86 Interrupt List (http://www.cs.cmu.edu/~ralf/files.html) (a.k.a. RBIL, Ralf Brown's Interrupt List)
- Embedded BIOS User's Manual (ftp://ftp.embeddedarm.com/old/saved-downloads-manuals/EBIOS-UM.PDF)
- PhoenixBIOS 4.0 User's Manual (http://www.esapcsolutions.com/ecom/drawings/PhoenixBIOS4_rev6UserMan.pdf)
- *IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference*, IBM, 1988, OCLC 20737442 (https://www.worldcat.org/oclc/20737442)
- *System BIOS for IBM PCs, Compatibles, and EISA Computers*, Phoenix Technologies, 1991, ISBN 0201577607
- *Programmer's Guide to the AMIBIOS*, American Megatrends, 1993, ISBN 0070015619
- *The Programmer's PC Sourcebook* by Thom Hogan, Microsoft Press, 1991 ISBN 155615321X