Know your Personal Computer

8. Basic Input - Output System (BIOS)

S K Ghoshal

Siddhartha Kumar Ghoshal works with whatever goes on inside parallel computers. That includes hardware, system software, algorithms and applications. From his early childhood he has designed and built electronic gadgets. One of the most recent ones is a sixteen processor parallel computer with IBM PC motherboards.

This article briefly describes the firmware organization of the IBM PC architecture.

Introduction

In part 2 of this series we saw that BIOS is the firmware of the IBM PC present in EPROMs plugged into every motherboard. We shall survey BIOS in this article. In 1981, when the nomenclature associated with the IBM PC were decided upon and standardized, one invoked operating systems mostly to do Input/Output operations. Operating systems, particularly the ones that ran on small computers those days were very primitive. They only provided certain services involving devices that had to be driven and/or handled. Thus many low-level operating systems that were designed to run on board-level products built in the early 80's were called Basic Input-Output Systems to inform the naive users that higher level operating systems were available to handle high-level and application-friendly I/O operations. As board-level system integration technology flourished, many hardware modules that enhance the architecture of the personal computer were added. One needed low-level operating systems to initialize and manage them, make them appear 'intelligent' (see part 3 of this series) or even to make the higher levels of system software to function in case they were absent or failed in a given motherboard. BIOS was upgraded to fit into that role. It retains its name(which severely understates its capabilities and role) and the set of all system calls which its ancestors provided. A survey of the BIOS system calls will reveal how capabilities to do powerful and high-level operations needed to run operating systems and application programs efficiently have percolated from the highest level of

system software down into the hardware and how a modern BIOS deftly controls that hardware giving you performance, flexibility, upward-compatibility and scalability. What needs hundreds of lines of C code in a high-level operating system implementation and takes hundreds of milliseconds to execute on a modern microprocessor running a primitive BIOS, gets done in a few hundred microseconds, by one system call to a modern BIOS running on the same CPU. Thus invoking BIOS directly from your application programs gives you a lot more speed and capabilities. You lose portability however, as I warned you in part 3 of this series. Modern BIOS can and does give you, right now, capabilities which the most modern high-level languages and operating system (even for operating systems specially developed for the PC architecture) calls do not support or even dream of supporting. And it all comes free (in the EPROM which is a part of the motherboard hardware, as seen in part 2 of this series) when you buy a modern motherboard. If you know how to use it, go ahead. Just one caveat. Modern BIOS has also inherited all the bad implementation practices which have been predominant in high-level operating systems. So accept the advanced BIOS calls with the same degree of scepticism with which you view system calls to high-level operating systems. When you use either of them, make sure that it is working as per its specification without causing some other unwanted side-effects.

The code and read-only data of BIOS are in the *Erasable Programmable Read Only Memory* (EPROM). The EPROM, by nature is a very slow memory, has only a 8-bit wide data path with the CPU and its contents cannot be selectively over-written. In addition to read-only memory, BIOS also needs writable memory to function, which it borrows from the RAM of the motherboards. Old BIOS (from now on old BIOS also includes the subset of the new that is compatible with the old) uses the low order addresses of the RAM. Modern BIOS occasionally expands the memory (see part 6 of this series) to cover its own EPROM. Modern BIOS can also copy itself into

The previous articles of this series were:

1. Introduction to computers, January 1996.
2. The personal computer hardware, February 1996.
3. The personal computer system software, April 1996.
4. The CPU base architecture, July 1996.
5. The CPU base instruction set and assembly language programming, November 1996.
6. Memory organisation, February 1997.
7. Input-Output ports, May 1997.

The debate on whether we should go for well-designed operating systems or the ones that work well on practical architectures continues. Surf the internet, beginning at the URL I have cited, to benefit from the debate.

RAMs which are connected to the CPU by a 32-bit wide data path thereby enhancing the performance of BIOS system calls. This technique is called *shadow BIOS*. The FSMs (see parts 2 and 6 of this series) are programmed to prevent the CPU from accidentally writing into the shadow. The control signal that indicates a write operation (see *Figure 1* in part 7) does not reach the write input pin of the RAM chip, if the address decodes into the shadow BIOS zone.

BIOS can be divided into two parts:

- The initialization routines, called POST (*Power-On Self Test*)
- Implementation of low-level system calls

These will be discussed in the sections that follow.

POST

Remember these facts:

- Before any module in the board-level hardware can be used, it must be initialized by POST. (See part 2 of this series.)

Accident?

"What kind of an accident can make BIOS to try to destroy itself? ", you may wonder. In fact had BIOS not been executed from Read-Only Memories (ROMs), it would have destroyed itself while trying to do its normal work. BIOS has many bugs. Some of them try to write into its code segment. As the ROM does not have the capability of being written into, these attempts fail and BIOS survives without anybody even suspecting anything going wrong. It is natural for BIOS to have such bugs. When some defect does not prevent you from using an equipment the way you want to use it, you don't bother about it. Also BIOS has become so huge in size that it is impossible to keep track of all its bugs. Old BIOS was written in assembly language (see part 5 of this series) and the technical reference manual of the original IBM PC listed that code with comments. People really knew how to write compact and well-documented programs then. However, even then BIOS had many bugs. Modern BIOS is written in C. It also has many bugs. No individual or a group of software developers would know about all the bugs that exist in any modern BIOS version.

- POST has several phases. At each phase, POST initializes complex modules using routines that run on a partially functional architecture made out of simple modules. In this mechanism, the on-board architecture, in several phases, becomes complete and attains all the capabilities that have been built into it till now.
- Working modules can indicate that they are working properly but damaged modules cannot indicate that they are damaged. Thus, at each phase, POST first tests the more reliable and crucial modules. Then it attempts to test those modules that are known to be failure-prone. Next it tests those modules without which it can still work and host the next phases.

The address from which an 80X86 microprocessor will begin execution, on being reset, is determined by the microprocessor architect and cannot be changed. The POST implementor must keep an EPROM at that address.

POST begins at an address **0xffff0** which is the restart address for the RESET signal of the 80X86 microprocessors. As this is almost at the end of the 1MB memory space addressable by 20 address bits, the first instruction is usually a jump to **0xfe05b**. The real POST begins here.

First, POST tests the microprocessor. Register to register moves, loads and stores, ALU operations, TLB functions, presence and health of On-chip cache and TAG RAM, and proper setting/resetting of flags are among the features tested at this stage. Any error causes the processor to enter the HALT state. These modules are very crucial. There is no point in going ahead with POST, if any of these fail.

Until POST comes to the stage where it can initialize and use the display controller to report the results of its diagnostics, it uses an I/O port (with address **0x60**) to report them. See part 7 of this series.

POST then carries out a crude test for the presence of primary memory. It just tests at 16K boundaries. Note here, that unless memory is operational, no instruction that relies on memory

A modulo 2^{16} checksum is computed by summing all the bytes of the BIOS EPROM in a 16-bit variable.

can be executed and POST has many such instructions. So before subprograms can be called, interrupts handled and status of any operation saved/restored, primary (read-write) memory must be usable.

POST then tests for any corruption in the BIOS EPROM.This is extremely important. During initialization, POST has to write bit-patterns into the different controller registers. An illegal pattern can potentially damage hardware permanently. So before dealing with any on-board controllers, POST ensures that the BIOS EPROM is not corrupted. This is done by computing a modulo 2^{16} checksum and verifying it against a stored value. The first BIOS from IBM had a checksum of **0x9a00**. It is customary to have all BIOS checksums as an integral multiple of **0x100**.

Timer controllers are initialized thereafter. One channel of the on-board programmable interval timer is programmed to generate the timer-tick interrupt. Each interrupt is handled by BIOS to do housekeeping chores that must be done periodically. Later on, higher level operating systems take over this responsibility.

Then the DMA controllers are initialized. On systems that rely on the DMA controller to refresh DRAMs, the DMA controller is programmed to refresh memory. Stack is set up at the top of the usable primary memory.

The interrupt controllers are initialized thereafter. Vectors for important hardware interrupts and BIOS traps are initialized to point out default handlers located within the BIOS EPROM.

The configuration is read from either the CMOS RAM or setting of jumpers/DIP switches or both. They are stored in the BIOS parameter block (BPB) area with a segment address of **0x40**. The bootstrap loader of higher level operating systems copies them into another area, called Boot-Kernel Interface (BKI).

The display controller modes and video memory are initialized. After that, detailed tests are performed on the timer, the interrupt controllers and the keyboard. Then a check is made for expansion boxes on the I/O expansion slots to see if they want to perform some initializations for the devices they represent.

Next a detailed test on the primary and extended memory are done. For 80386 and 80486 microprocessor based motherboards to address beyond 1MB, the processor switches onto a simple protected mode at this stage and comes back to real mode once it is over.

Then a check is made for the presence of optional ROMs in the address range **0xc8000** to **0f4000**. These ROMs can take over the thread of control from POST, to perform some special functions, like BASIC interpretation, running of special-purpose monitors etc.

A test is made for the presence, health and number of the secondary devices present in the system. 1.2MB (5.25 inch) floppies, 1.44MB (3.5 inch) floppies, and hard disks ranging from type 2 to type 47 are checked. Drives are recalibrated.

Printer and serial ports are then tested. BPB is updated by their number and status.

The POST calls the BIOS bootstrap function by invoking the firmware trap **INT 19H**. This reads the boot sector from the boot device and loads the first 512 bytes into the memory area beginning at **0x7c00**. POST then transfers control to that address. The bootstrap loader of the operating system then takes over. We shall discuss this in future articles.

BIOS System Calls

These are implemented as assembly language routines and kept in the BIOS EPROM. POST initializes the restart vectors of

Figure 1 A program to invoke BIOS

```
MEMSIZ PROC NEAR; Use a small memory model. See part 5.
INT 12H; Invoke BIOS interrupt to size memory
RET ; Go back to caller with value in AX
MEMSIZ ENDP; The assembly language procedure ends here
```

certain reserved interrupts to point to them. Once the thread of control is inside the EPROM, the value of the AH register (see part 4 of this series) is tested. Depending on this value, the control jumps to different inside routines of BIOS implementation. This is similar to the way a **case** statement in Pascal or a **switch** statement in C works. A single restart location serves as the entry point for a large number of BIOS routines implementing features of a given *genre* that is associated with the interrupt. To invoke the BIOS routine, execute an INT instruction. Parameters are passed between BIOS and the caller using registers of the 80X86. For an example see *Figure 1*. This is an assembly language caller program of BIOS. It invokes INT 12H which makes BIOS return the size of primary (i.e. non-expanded, non-extended, below 640KB as in *Figure 2* of part 6) memory present on the motherboard, expressed in Kilobytes. The program in *Figure 1* is itself C-callable in turn and in part 5 of this series we saw how to link and run it with your own C caller. Note:

- INT 12H provides only one service. Thus we need not initialize AH in *Figure 1*.
- INT 12H returns its result in the AXregister. We just pass that on to the C caller. See part 5 of this series.

In subsequent articles, we shall discuss in detail the role of higher level operating systems and how they are started up. If you have any reaction to this article, write to me.

Address for Correspondence
S K Ghoshal
Supercomputer Education and Research Centre
Indian Institute of Science
Bangalore 560 012, India
email:
ghoshal@serc.iisc.ernet.in
Fax: (080) 334 1683

Suggested Reading

- ***The IBM PC/AT Technical Reference Manual.*** **IBM Corporation, 1985.**
- **http://www.dina.kvl.dk/~abraham/Linus_Vs_Tanenbaum.html.**