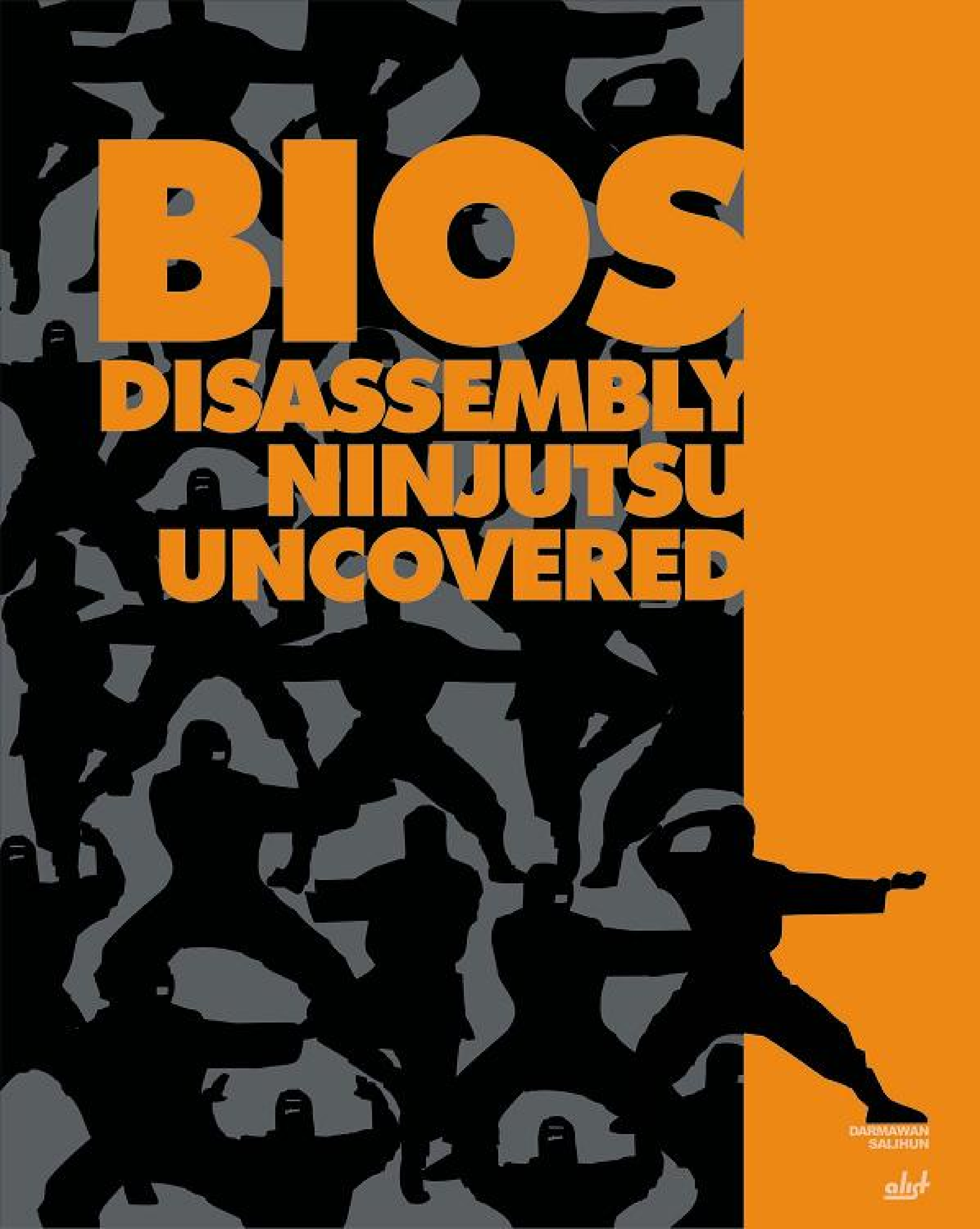
# BIOS DISASSEMBLY NINJUTSU UNCOVERED



DARMAWAN SALIHUN

alist

# Preface

## BIOS DISASSEMBLY NINJUTSU UNCOVERED – THE BOOK

For many years, there has been a myth among computer enthusiasts and practitioners that PC BIOS (Basic Input Output System) modification is a kind of black art and only a handful of people can do it or only the motherboard vendor can carry out such a task. On the contrary, this book will prove that with the right tools and approach, anyone can understand and modify the BIOS to suit their needs without the existence of its source code. It can be achieved by using a systematic approach to BIOS reverse engineering and modification. An advanced level of this modification technique is injecting a custom code to the BIOS binary.

There are many reasons to carry out BIOS reverse engineering and modification, from the fun of doing it to achieve higher clock speed in overclocking scenario, patching certain bug, injecting a custom security code into the BIOS, up to commercial interest in the embedded x86 BIOS market. The emergence of embedded x86 platform as consumer electronic products such as TV set-top boxes, telecom-related appliances and embedded x86 kiosks have raised the interest in BIOS reverse engineering and modification. In the coming years, these techniques will become even more important as the state of the art bus protocols have delegate a lot of their initialization task to the firmware, i.e. the BIOS. Thus, by understanding the techniques, one can dig the relevant firmware codes and understand the implementation of those protocols within the BIOS binary.

The main purpose of the BIOS is to initialize the system into execution environment suitable for the operating system. This task is getting more complex over the years, since x86 hardware evolves quite significantly. It's one of the most dynamic computing platform on earth. Introduction of new chipsets happens once in 3 or at least 6 month. This event introduces a new code base for the silicon support routine within the BIOS. Nevertheless, the overall architecture of the BIOS is changing very slowly and the basic principle of the code inside the BIOS is preserved over generations of its code. However, there has been a quite significant change in the BIOS scene in the last few years, with the introduction of EFI (extensible Firmware Interface) by several major hardware vendors and with the growth in OpenBIOS project. With these advances in BIOS technology, it's even getting more important to know systematically what lays within the BIOS.

In this book, the term BIOS has a much broader meaning than only motherboard BIOS, which is familiar to most of the reader. It also means the expansion ROM. The latter term is the official term used to refer to the firmware in the expansion cards within the PC, be it ISA, PCI or PCI Express.

So, what can you expect after reading this book? Understanding the BIOS will open a new frontier. You will be able to grasp how exactly the PC hardware works in its lowest level. Understanding contemporary BIOS will reveal the implementation of the latest bus protocol technology, i.e. HyperTransport and PCI-Express. In the software engineering front, you will be able to appreciate the application of compression technology in the BIOS. The most important of all, you will be able to carry out reverse engineering using advanced techniques and tools. You will be able to use the powerful IDA Pro disassembler efficiently. Some reader with advanced knowledge in hardware and software might even want to "borrow" some of the algorithm within the BIOS for their own purposes. In short, you will be on the same level as other BIOS code-diggers.

This book also presents a generic approach to PCI expansion ROM development using the widely available GNU tools. There will be no more myth in the BIOS and everyone will be able to learn from this state-of-the-art software technology for their own benefits.

## THE AUDIENCE

This book is primarily oriented toward system programmers and computer security experts. In addition, electronic engineers, pc technicians and computer enthusiasts can also benefit a lot from this book. Furthermore, due to heavy explanation of applied computer architecture (x86 architecture) and compression algorithm, computer science students might also find it useful. However, nothing prevents any people who is curious about BIOS technology to read this book and get benefit from it.

Some prerequisite knowledge is needed to fully understand this book. It is not mandatory, but it will be very difficult to grasp some of the concepts without it. The most important knowledge is the understanding of x86 assembly language. Explanation of the disassembled code resulting from the BIOS binary and also the sample BIOS patches are presented in x86 assembly language. They are scattered throughout the book. Thus, it's vital to know x86 assembly language, even with very modest familiarity. It's also assumed that the reader have some familiarity with C programming language. The chapter that dwell on expansion ROM development along with the introductory chapter in BIOS related software development uses C language heavily for the example code. C is also used heavily in the section that covers IDA Pro scripts and plugin development. IDA Pro scripts have many similarities with C programming language. Familiarity with Windows Application Programming Interface (Win32API) is not a requirement, but is very useful to grasp the concept in the Optional section of chapter 3 that covers IDA Pro plugin development.

## THE ORGANIZATION

The first part of the book lays the foundation knowledge to do BIOS reverse engineering and Expansion ROM development. In this part, the reader is introduced with:

a. Various bus protocols in use nowadays within the x86 platform, i.e. PCI, HyperTransport and PCI-Express. The focus is toward the relationship between BIOS code execution and the implementation of protocols.
b. Reverse engineering tools and techniques needed to carry out the tasks in later chapter, mostly introduction to IDA Pro disassembler along with its advanced techniques.
c. Crash course on advanced compiler tricks needed to develop firmware. The emphasis is in using GNU C compiler to develop a firmware framework.

The second part of this book reveals the details of motherboard BIOS reverse engineering and modification. This includes indepth coverage of BIOS file structure, algorithms used within the BIOS, explanation of various BIOS specific tools from its corresponding vendor and explanation of tricks to perform BIOS modification.

The third part of the book deals with the development of PCI expansion ROM. In this part, PCI Expansion ROM structure is explained thoroughly. Then, a systematic PCI expansion ROM development with GNU tools is presented.

The fourth part of the book deals heavily with the security concerns within the BIOS. This part is biased toward possible implementation of rootkits within the BIOS and possible exploitation scenario that might be used by an attacker by exploiting the BIOS flaw. Computer security experts will find a lot of important information in this part. This part is the central theme in this book. It's presented to improve the awareness against malicious code that can be injected into BIOS.

The fifth part of the book deals with the application of BIOS technology outside of its traditional space, i.e. the PC. In this chapter, the reader is presented with various application of the BIOS technology in the emerging embedded x86 platform. In the end of this part, further application of the technology presented in this book is explained briefly. Some explanation regarding the OpenBIOS and Extensible Firmware Interface (EFI) is also presented.

## SOFTWARE TOOLS COMPATIBILITY

This book mainly deals with reverse engineering tools running in windows operating system. However, in chapters that deal with PCI Expansion ROM development, an x86 Linux installation is needed. This is due to the inherent problems that occurred with the windows port of the GNU tools when trying to generate a flat binary file from ELF file format.

# Proposed Table of Contents

# Typographical Conventions

In this book, the `courier` font is used to indicate that text is one of the following:

1. Source code
2. Numeric values
3. Configuration file entries
4. Directory/paths in the file system
5. Datasheet snippets
6. CPU registers

Hexadecimal values are indicated by prefixing them with a `0x or by` appending them with h. For example, the integer value `4691` will, in hexadecimal, look like `0x1253 or 1253h`. Hexadecimal values larger than four digits will be accompanied by underscore every four consecutive hexadecimal digits to ease reading the value, as in `0xFFFF_0000` and `0xFD_FF00_0000`.

Binary values are indicated by appending them with b. For example, the integer value `5` will, in binary, look like `101b`.

Words will appear in the *italic* font, in this book, for following reasons:

1. When defining a new term
2. For emphasis

Words will appear in the **bold** font, in this book, for the following reasons:

3. When describing a menu within an application software in Windows
4. A key press, e.g. **CAPSLOCK**, **G**, **Shift**, **C**, etc.
5. For emphasis

# *Part I The Basics*

# *Chapter 1 PC BIOS Technology*

## *PREVIEW*

This chapter is devoted to explaining the parts of a PC that make up the term basic input/output system (BIOS). These are not only motherboard BIOS, which most readers might already be accustomed to, but also expansion read-only memories (ROMs). The BIOS is one of the key parts of a PC. BIOS provides the necessary execution environment for the operating system. The approach that I take to explain this theme follows the logic of the execution of BIOS subsystems inside the PC. It is one of the fastest ways to gain a systematic understanding of BIOS technology. In this journey, you will encounter answers to common questions: Why is it there? Why does it have to be accomplished that way? The discussion starts with the most important BIOS, motherboard BIOS. On top of that, this chapter explains contemporary bus protocol technology, i.e., PCI Express, HyperTransport, and peripheral component interconnect (PCI). A profound knowledge of bus protocol technology is needed to be able to understand most contemporary BIOS code.

## 1.1. Motherboard BIOS

Motherboard BIOS is the most widely known BIOS from all kinds of BIOS. This term refers to the machine code that resides in a dedicated ROM chip on the motherboard. Today, most of these ROM chips are the members of flash-ROM family. This name refers to a ROM chip programmed[1] electrically in a short interval, i.e., the programming takes only a couple of seconds.

There is a common misconception between the BIOS chip and the complementary metal oxide semiconductor (CMOS) chip. The former is the chip that's used to store the *BIOS code*, i.e., the machine code that will be executed when the processor executes the BIOS, and the latter is the chip that's used to store the *BIOS parameters*, i.e., the parameters that someone sets when entering the BIOS, such as the computer date and the RAM timing. Actually, CMOS chip is a misleading name. It is true that the chip is built upon CMOS technology. However, the purpose of the chip is to store BIOS information with the help of a dedicated battery. In that respect, it should've been called non-volatile random access memory (NVRAM) chip in order to represent the nature and purpose of the chip. Nonetheless, the *CMOS chip* term is used widely among PC users and hardware vendors.

[1] Programmed in this context means being erased or written into.

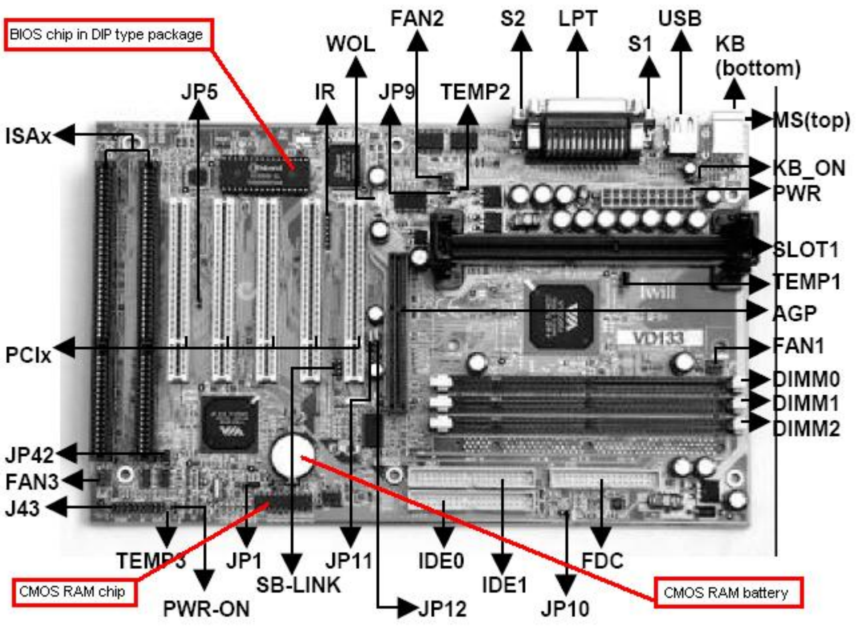


**Figure 1.1 Motherboard with a DIP-type BIOS chip**

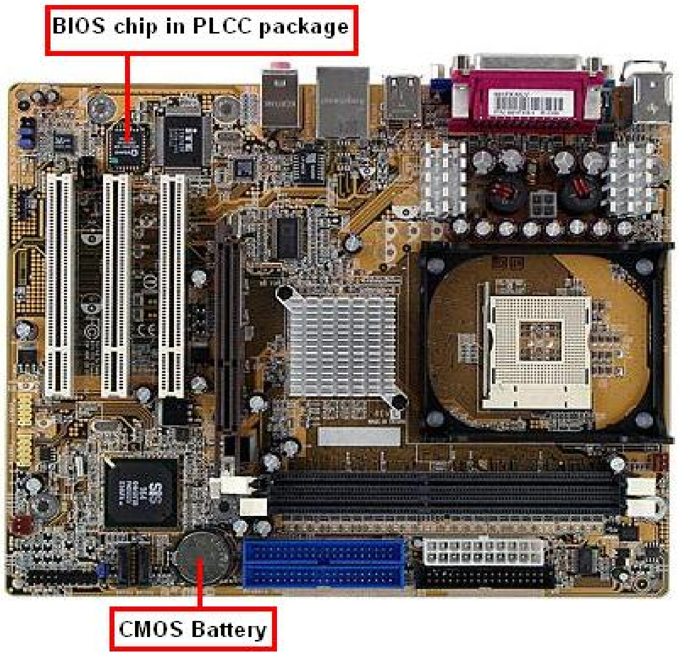


**Figure 1.2 Motherboard with a PLCC-type BIOS chip**

The widely employed chip packaging for BIOS ROM is PLCC[2] (fig. 1.1) or DIP[3] (fig. 1.2). Modern-day motherboards mostly use the PLCC package type. The top marking on the BIOS chip often can be seen only after the BIOS vendor sticker, e.g., Award BIOS or AMI BIOS, is removed. The commonly used format is shown in figure 1.3.

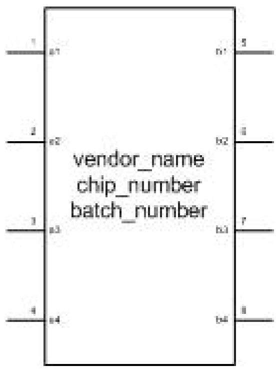


**Figure 1.3 BIOS chip marking**

1. The `vendor_name` is the name of the chip vendor, such as Winbond, SST, or Atmel.
2. The `chip_number` is the part number of the chip. Sometimes this part number includes the access time specification of the corresponding chip.
3. The `batch_number` is the batch number of the chip. It is used to mark the batch in which the chip belonged when it came out of the factory. Some chips might have no batch number.

This chip marking is best explained by using an example (fig. 1.4).

[2] Plastic lead chip carrier, one of the chip packaging technologies.
[3] Dual inline package, one of the chip packaging technologies.

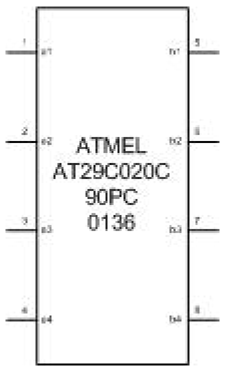


**Figure 1.4 BIOS chip marking example**

In the marking in figure 1.4, the AT prefix means "made by Atmel," the part number is 29C020C, and 90PC means the chip has 90 ns of access time. Detailed information can be found by downloading and reading the datasheet of the chip from the vendor's website. The only information needed to obtain the datasheet is the part number.

It is important to understand the BIOS chip marking, especially the part number and the access time. The access time information is always specified in the corresponding chip datasheet. This information is needed when you intend to back up your BIOS chip with a chip from a different vendor. The access time and voltage level of both chips must match. Otherwise, the backup process will fail. The backup process can be carried out by hot swapping or by using specialized tools such as BIOS Saviour. Hot swapping is a dangerous procedure and is *not* recommended. Hot swapping can destroy the motherboard and possibly another component attached to the motherboard if it's not carried out carefully. However, if you are adventurous, you might want to try it in an old motherboard. The hot swapping steps are as follows:

1. Prepare a BIOS chip with the same type as the one in the current motherboard to be used as the target, i.e., the new chip that will be flashed with the BIOS in the current motherboard. This chip will act as the BIOS backup chip. Remove any sticker that keeps you from seeing the type of your motherboard BIOS chip (usually the Award BIOS or AMI BIOS logo). This will void your motherboard warranty, so proceed at your own risk. The same type of chip here means a chip that has the same part number as the current chip. If one can't be found, you can try a compatible chip, i.e., a chip that has the same capacity, voltage level, and timing characteristic. Note that finding a compatible chip is not too hard. Often, the vendor of flash-ROMs provides flash-ROM cross-reference documentation in their website. This documentation lists the compatible flash-ROM from other vendors. Another way to find a compatible chip is to download datasheets from two different vendors with similar part numbers and compare their properties according to both datasheets. If the capacity, voltage level, and access time match, then the chip is compatible. For example, ATMEL AT29C020C is compatible with WINBOND W29C020C.

2. Prepare the BIOS flashing software in a diskette or in a file allocation table (FAT) formatted hard disk drive (HDD) partition. This software will be used to save BIOS binary from the original BIOS chip and to flash the binary into the backup chip. The BIOS flashing software is provided by the motherboard maker from its website, or sometimes it's shipped with the motherboard driver CD.
3. Power off the system and unplug it from electrical source. Loosen the original BIOS chip from the motherboard. It can be accomplished by first removing the chip using a screwdriver or IC extractor from the motherboard and then reattaching it firmly. Ensure that the chip is not attached too tightly to the motherboard and it can be removed by hand later. Also, ensure that electrical contact between the IC and the motherboard is strong enough so that the system will be able to boot.
4. Boot the system to the real-mode disk operating system (DOS). Beware that some motherboards may have a BIOS flash protection option in their BIOS setup. It has to be disabled before proceeding to the next step.
5. Run the BIOS flashing software and follow its on-screen direction to save the original BIOS binary to a FAT partition in the HDD or to a diskette.
6. After saving the original BIOS binary, carefully release the original BIOS chip from the motherboard. Note that this procedure is carried out with the computer still running in real-mode DOS.
7. Attach the backup chip to the motherboard firmly. Ensure that the electrical contact between the chip and the motherboard is strong enough.
8. Use the BIOS flashing software to flash the saved BIOS binary from the HDD partition or the diskette to the backup BIOS chip.
9. Reboot the system and see whether it boots successfully. If it does, the hot swapping has been successful.

Hot swapping is not as dangerous as you might think for an experienced hardware hacker. Nevertheless, use of a specialized device such as BIOS Saviour for BIOS backup is bulletproof.

Anyway, you might ask, why would the motherboard need a BIOS? There are several answers to this seemingly simple question. First, system buses, such as PCI, PCI-X, PCI Express, and HyperTransport consume memory address space and input/output (I/O) address space. Devices that reside in these buses need to be initialized to a certain address range within the system memory or I/O address space before being used. Usually, the memory address ranges used by these devices are located above the address range used for system random access memory (RAM) addressing. The addressing scheme depends on the motherboard chipset. Hence, you must consult the chipset datasheet(s) and the corresponding bus protocol for details of the addressing mechanism. I will explain this issue in a later chapter that dwells on the bus protocol.

Second, some components within the PC, such as RAM and the central processing unit (CPU) are running at the "undefined" clock speed[4] just after the system is powered up. They must be initialized to some predefined clock speed. This is where the BIOS comes into play; it initializes the clock speed of those components.

The bus protocol influences the way the code inside the BIOS chip is executed, be it motherboard BIOS or other kinds of BIOS. Section 1.4 will delve into bus protocol fundamentals to clean up the issue.

## 1.2. Expansion ROM

Expansion ROM[5] is a kind of BIOS that's embedded inside a ROM chip mounted on an add-in card. Its purpose is to initialize the board in which it's soldered or socketed before operating system execution. Sometimes it is mounted into an old ISA add-in card, in which case it's called ISA expansion ROM. If it is mounted to a PCI add-in card, it's called PCI expansion ROM. In most cases, PCI or ISA expansion ROM is implanted inside an erasable or electrically erasable programmable read-only memory chip or a flash-ROM chip in the PCI or ISA add-in card. In certain cases, it's implemented as the motherboard BIOS component. Specifically, this is because of motherboard design that incorporates some onboard PCI chip, such as a redundant array of independent disks (RAID) controller, SCSI controller, or serial advanced technology attachment (ATA) controller. Note that expansion ROM implemented as a motherboard BIOS component is no different from expansion ROM implemented in a PCI or ISA add-in card. In most cases, the vendor of the corresponding PCI chip that needs chip-specific initialization provides expansion ROM binary. You are going to learn the process of creating such binary in part 3 of this book.

[4] *"Undefined" clock speed* in this context means the power-on default clock speed.
[5] *Expansion ROM* is also called as *option ROM* in some articles and documentations. The terms are interchangeable.

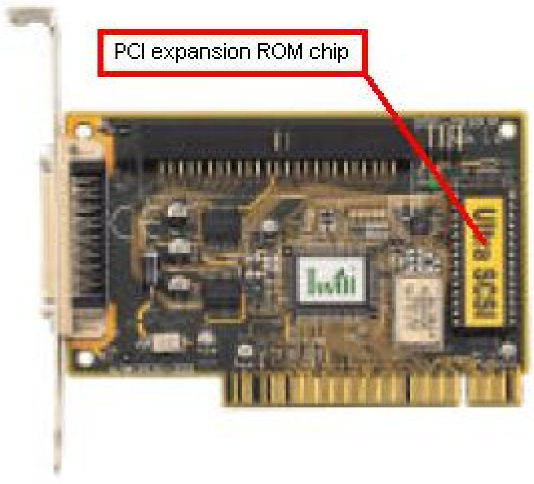


**Figure 1.5 PCI expansion ROM chip**

Actually, there is some complication regarding PCI expansion ROM execution compared with ISA expansion ROM execution. ISA expansion ROM is executed in place,[6] and PCI expansion ROM is always copied to RAM and executed from there. This issue will be explained in depth in the chapter that covers the PCI expansion ROM.

## 1.3. Other Firmware within the PC

It must be noted that motherboard and add-in cards are not the only ones that possess firmware. HDDs and CD-ROM drives also possess firmware. The firmware is used to control the physical devices within those drives and to communicate with the rest of the system. However, those kinds of firmware are not considered in this book. They are mentioned here just to ensure that you are aware of their existence.

## 1.4. Bus Protocols Fundamentals

This section explains bus protocols used in a PC motherboard, namely PCI, PCI Express, and HyperTransport. These protocols are tightly coupled with the BIOS. In fact, the BIOS is part of the bus protocol implementation. The BIOS handles the initialization of the addressing scheme employed in these buses. The BIOS handles another protocol-specific initialization. This section is not a thorough explanation of the bus protocols

[6] *Executed in place* means executed from the ROM chip in the expansion card.

themselves; it is biased toward BIOS implementation-related issues, particularly the *programming model* employed in the respective bus protocol.

First, it delves into the system-wide addressing scheme in contemporary systems. This role is fulfilled by the chipset. Thus, a specific implementation is used as an example.

### 1.4.1. System-wide Addressing

If you have never been playing around with system-level programming, you might find it hard to understand the organization of the overall physical memory address space in x86 architecture. It must be noted that *RAM is not the only hardware that uses the processor memory address space*; some other hardware is also mapped to the processor memory address space. This memory-mapped hardware includes PCI devices, PCI Express devices, HyperTransport devices, the advanced programmable interrupt controller (APIC), the video graphics array (VGA) device, and the BIOS ROM chip. It's the responsibility of the chipset to divide the x86 processor memory address space for RAM and other memory-mapped hardware devices. Among the motherboard chipsets, the northbridge is responsible for this system address-space organization, particularly its memory controller part. The memory controller decides where to forward a read or write request from the CPU to a certain memory address. This operation can be forwarded to RAM, memory-mapped VGA RAM, or the southbridge; it depends on the system configuration. If the northbridge is embedded inside the CPU itself, like in the AMD Athlon 64/Opteron architecture, the CPU decides where to forward these requests.

The influence of the bus protocol employed in x86 architecture to the system address map is enormous. To appreciate this, analyze a sample implementation in the form of a PCI Express chipset, Intel 955X-ICH7(R). This chipset is used with Intel Pentium 4 processors that support IA-32E and are capable of addressing RAM above the 4-GB limit.

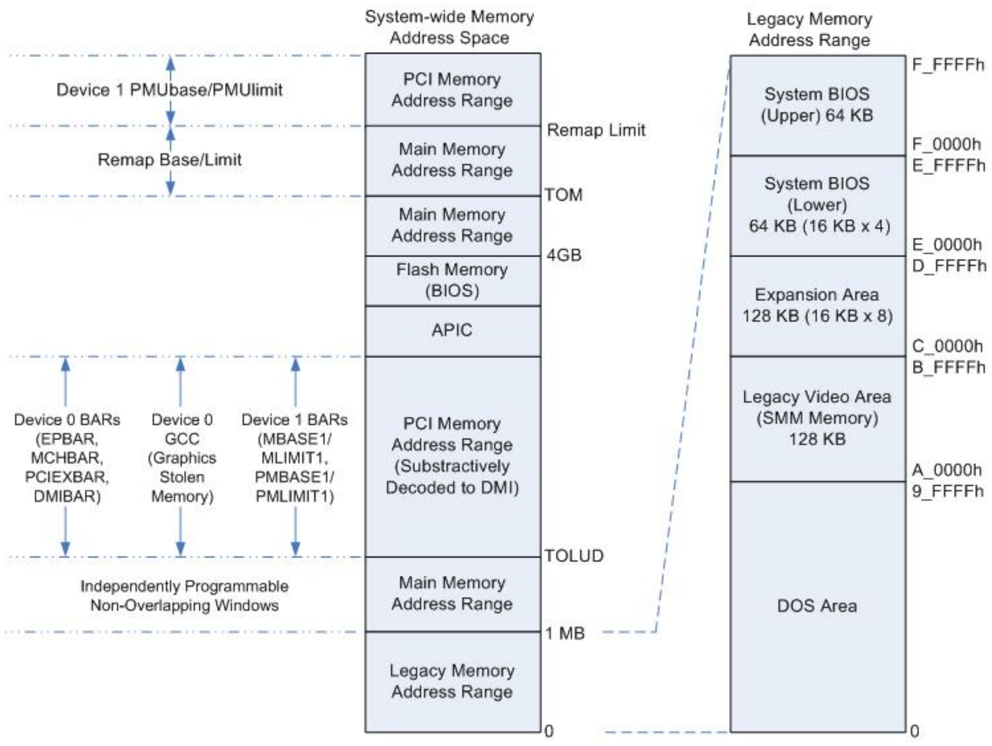

System-wide Memory Address Space
Legacy Memory Address Range
Device 1 PMUbase/PMUlimit
Remap Base/Limit
PCI Memory Address Range
Main Memory Address Range
Remap Limit
TOM
Main Memory Address Range
4GB
Flash Memory (BIOS)
APIC
Device 0 BARs (EPBAR, MCHBAR, PCIEXBAR, DMIBAR)
Device 0 GCC (Graphics Stolen Memory)
Device 1 BARs (MBASE1/ MLIMIT1, PMBASE1/ PMLIMIT1)
PCI Memory Address Range (Substractively Decoded to DMI)
TOLUD
Independently Programmable Non-Overlapping Windows
Main Memory Address Range
1 MB
Legacy Memory Address Range
0
System BIOS (Upper) 64 KB
System BIOS (Lower) 64 KB (16 KB x 4)
Expansion Area 128 KB (16 KB x 8)
Legacy Video Area (SMM Memory) 128 KB
DOS Area
F_FFFFh
F_0000h
E_FFFFh
E_0000h
D_FFFFh
C_0000h
B_FFFFh
A_0000h
9_FFFFh
0

**Figure 1.6 Intel 955X-ICH7(R) system address map**

Figure 1.6 shows that memory address space above the physical RAM is used for PCI devices, APIC, and BIOS flash-ROM. In addition, there are two areas of physical memory address space used by the RAM, i.e., below and above the 4-GB limit. This division is the result of the 4-GB limit of 32-bit addressing mode of x86 processors. Note that PCI Express devices are mapped to the same memory address range as PCI devices but they can't overlap each other. Several hundred kilobytes of the RAM address range is not addressable because its address space is consumed by other memory-mapped hardware devices, though this particular area may be available through system management mode (SMM). This is because of the need to maintain compatibility with DOS. In the DOS days, several areas of memory below 1 MB (`10_0000h`) were used to map hardware devices, such as the video card buffer and BIOS ROM. The "BARs" mentioned in figure 1.6 are an abbreviation for base address registers. These will be explained in a later section.

The system address map in figure 1.6 shows that the BIOS chip is mapped to two different address ranges, i.e., `4GB_minus_BIOS_chip_size` to 4 GB and `E_0000h` to `F_FFFFh`. The former BIOS flash-ROM address range varies from chipset to chipset, depending on the maximum BIOS chip size supported by the chipset. This holds true for *every* chipset and must be taken into account when I delve into the BIOS code in later chapters. The latter address range mapping is supported in most contemporary chipsets. This 128-KB range (`E_0000h`–`F_FFFFh`) is an alias to the topmost 128-KB address range in the BIOS chip. Chipsets based on a different bus protocol, such as HyperTransport or the older chipsets based on PCI, also employ mapping of physical memory address space similar to that described here. It has to be done that way to maintain compatibility with the current BIOS code from different vendors and to maintain compatibility with legacy software. Actually, there are cost savings in employing this addressing scheme; the base code for the BIOS from all BIOS vendors (AMI, Award Phoenix, etc.) need not be changed or only needs to undergo minor changes.

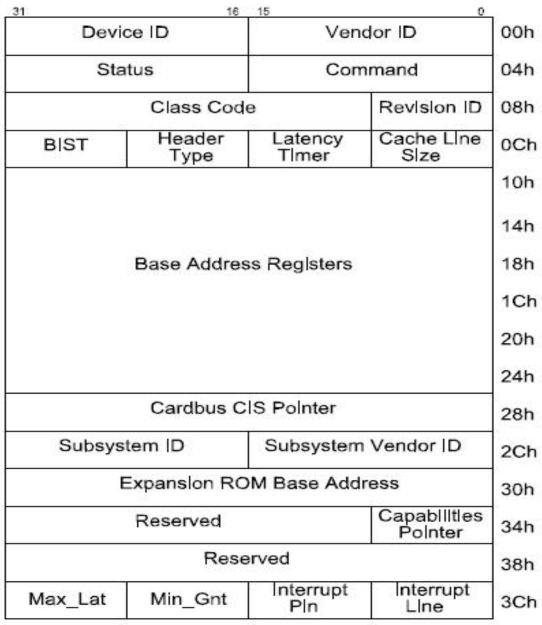
### 1.4.2. PCI Bus Protocol

The PCI bus is a high-performance 32-bit or 64-bit parallel bus with multiplexed address and data lines. The bus is intended for use as an interconnect mechanism between highly integrated peripheral controller components, peripheral add-in cards, and processor or memory systems. It is the most widely used bus in PC motherboard design since the mid-1990s. It's only recently that this bus system has been replaced by newer serial bus protocols, i.e., PCI Express and HyperTransport. The PCI Special Interest Group is the board that maintains the official PCI bus standard.

PCI supports up to 256 buses in one system, with every bus supporting up to 32 devices and every device supporting up to eight functions. The PCI protocol defines the so-called PCI-to-PCI bridges that connect two different PCI bus segments. This bridge forwards PCI transactions from one bus to the neighboring bus segment. Apart from extending the bus topology, the presence of PCI-to-PCI bridges is needed due to an electrical loading issue. The PCI protocol uses reflected-wave signaling that only enables around 10 onboard devices per bus or only five PCI connectors per bus. PCI connectors are used for PCI expansion cards, and they account for two electrical loads, one for the connector itself and one for the expansion card inserted into the connector.

The most important issue to know in PCI bus protocol with regard to BIOS technology is its programming model and configuration mechanism. This theme is covered in chapter 6 of the official PCI specification, versions 2.3 and 3.0. It will be presented with in-depth coverage in this section.

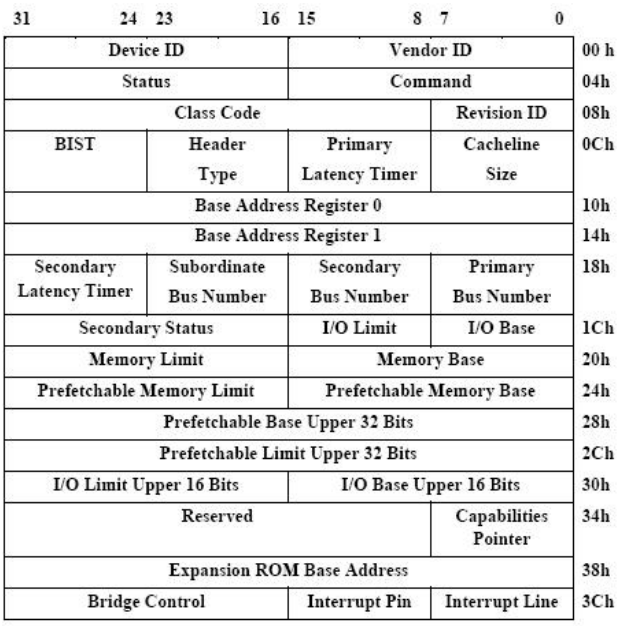
The PCI bus configuration mechanism is accomplished by defining 256-byte registers called *configuration space* in each logical PCI device function. Note that each physical PCI device can contain *more than one* logical PCI device and each logical device can contain *more than one* function. The PCI bus protocol doesn't specify a single mechanism used to access this configuration space for PCI devices in all processor architectures; on the contrary, each processor architecture has its own mechanism to access the PCI configuration space. Some processor architectures map this configuration space into their memory address space (memory mapped), and others map this configuration space into their I/O address space (I/O mapped). Figure 1.7 shows a typical PCI configuration space organization for PCI devices that's not a PCI-to-PCI bridge.



| 31 | 16 | 15 | 0 | |
|---|---|---|---|---|
| Device ID | | Vendor ID | | 00h |
| Status | | Command | | 04h |
| Class Code | | | Revision ID | 08h |
| BIST | Header Type | Latency Timer | Cache Line Size | 0Ch |
| Base Address Registers | | | | 10h |
| | | | | 14h |
| | | | | 18h |
| | | | | 1Ch |
| | | | | 20h |
| | | | | 24h |
| Cardbus CIS Pointer | | | | 28h |
| Subsystem ID | | Subsystem Vendor ID | | 2Ch |
| Expansion ROM Base Address | | | | 30h |
| Reserved | | | Capabilities Pointer | 34h |
| Reserved | | | | 38h |
| Max_Lat | Min_Gnt | Interrupt Pin | Interrupt Line | 3Ch |

**Figure 1.7 PCI configuration space registers for a non-PCI-to-PCI bridge device**

The PCI configuration space in x86 architecture is mapped into the processor I/O address space. The I/O port addresses `0xCF8`–`0xCFB` act as the *configuration address port* and I/O ports `0xCFC`–`0xCFF` act as the *configuration data port*. These ports are used to configure the corresponding PCI chip, i.e., reading or writing the PCI chip configuration register values. *It must be noted that the motherboard chipset itself, be it northbridge or southbridge, is a PCI chip. Thus, the PCI configuration mechanism is employed to configure these chips.* In most cases, these chips are a combination of several PCI functions or devices; the northbridge contains the host bridge, PCI–PCI bridge (PCI–accelerated graphics port bridge), etc., and the southbridge contains the integrated drive electronics controller, low pin count (LPC) bridge, etc. The PCI–PCI bridge is defined to address the electrical loading issue that plagues the physical PCI bus. In addition, recent bus architecture uses it as a logical means to connect different chips, meaning it's used to travel the bus topology and to configure the overall bus system. The typical configuration space register for a PCI–PCI bridge is shown in figure 1.8



| 31 24 | 23 16 | 15 8 | 7 0 | |
|---|---|---|---|---|
| Device ID | | Vendor ID | | 00 h |
| Status | | Command | | 04h |
| Class Code | | | Revision ID | 08h |
| BIST | Header Type | Primary Latency Timer | Cacheline Size | 0Ch |
| Base Address Register 0 | | | | 10h |
| Base Address Register 1 | | | | 14h |
| Secondary Latency Timer | Subordinate Bus Number | Secondary Bus Number | Primary Bus Number | 18h |
| Secondary Status | | I/O Limit | I/O Base | 1Ch |
| Memory Limit | | Memory Base | | 20h |
| Prefetchable Memory Limit | | Prefetchable Memory Base | | 24h |
| Prefetchable Base Upper 32 Bits | | | | 28h |
| Prefetchable Limit Upper 32 Bits | | | | 2Ch |
| I/O Limit Upper 16 Bits | | I/O Base Upper 16 Bits | | 30h |
| Reserved | | | Capabilities Pointer | 34h |
| Expansion ROM Base Address | | | | 38h |
| Bridge Control | | Interrupt Pin | Interrupt Line | 3Ch |

**Figure 1.8 PCI configuration space registers for a PCI-to-PCI bridge device**

Since the PCI bus is a 32-bit bus, communicating using this bus should be in 32-bit *addressing mode*. Writing or reading to this bus will require 32-bit addresses. Note that a 64-bit PCI bus is implemented by using *dual address cycle*, i.e., two address cycles are needed to access the address space of 64-bit PCI device(s). Communicating with the PCI configuration space in x86 is accomplished with the following algorithm (from the host or CPU point of view):

1. Write the target bus number, device number, function number, and offset or register number to the configuration address port (I/O ports `0xCF8`–`0xCFB`) and set the enable bit in it to one. In plain English: Write the address of the register that will be read or written into the PCI address port.
2. Perform a 1-byte, 2-byte, or 4-byte I/O read from or write to the configuration data port (I/O port `0xCFC`–`0xCFF`). In plain English: Read or write the data into the PCI data port.

With the preceding algorithm, you'll need an x86 assembly code snippet that shows how to use those configuration ports.

**Listing 1.1 PCI Configuration Read and Write Routine Sample**

```
; Mnemonic is in MASM syntax
  pushad       ; Save all contents of general-purpose registers.
```

```
mov eax,80000064h  ; Put the address of the PCI chip register to be
                   ; accessed in eax (offset 64 device 00:00:00 or
                   ; host bridge/northbridge).

mov dx,0CF8h       ; Put the address port in dx. Since this is PCI,
                   ; use 0xCF8 as the port to open access to
                   ; the device.
out dx,eax         ; Send the PCI address port to the I/O space of
                   ; the processor.

mov dx,0CFCh       ; Put the data port in dx. Since this is PCI,
                   ; use 0xCFC as the data port to communicate with
                   ; the device.

in eax,dx          ; Put the data read from the device in eax.

or eax, 00020202   ; Modify the data (this is only an example; don't
                   ; try this in your machine, it may hang or
                   ; even destroy your machine).

out dx,eax         ; Send it back

; ...              ; your routine here.

popad              ; Restore all the saved register.

ret                ; Return to the calling procedure.
```

This code snippet is a procedure that I injected into the BIOS of a motherboard based on a VIA 693A-596B PCI chipset to patch its memory controller configuration a few years ago. The code is clear enough; in line 1 the current data in the processor's general-purpose registers were saved. Then comes the crucial part, as I said earlier: PCI is a 32-bit bus system; hence, you have to use 32-bit addresses to communicate with the system. You do this by sending the PCI chip a 32-bit address through `eax` register and using port `0xCF8` as the port to send this data. Here's an example of the PCI register (sometimes called the *offset*) address format. In the routine in listing 1.1, you see the following:

```
...
mov eax,80000064h
...
```

The `80000064h` is the address. The meanings of these bits are as follows:

| **Bit Position** | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Binary Value** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| **Hexadecimal Value** | 0 | | | | 0 | | | | 6 | | | | 4 | | | |

**Figure 1.9 PCI configuration address sample (low word)**

| Bit Position | 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Binary Value | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Hexa-decimal Value | 8 | | | | 0 | | | | 0 | | | | 0 | | | |

**Figure 1.10 PCI configuration address sample (high word)**

| Bit Position | Meaning |
|---|---|
| 31 | This is an enable bit. Setting this bit to one will grant a write or read transaction through the PCI bus; otherwise, the transaction is not a valid configuration space access and it is ignored. That's why you need an `8` (`8h`) in the leftmost hex digit. |
| 24–30 | Reserved bits |
| 16–23 | *PCI bus number* |
| 11–15 | *PCI device number* |
| 8–10 | *PCI function number* |
| 2–7 | *Offset address* (double word or 32-bit boundary) |
| 0–1 | Unused, since the addressing must be in the 32-bit boundary |

**Table 1.1 PCI register addressing explanation**

Now, examine the previous value that was sent. If you are curious, you'll find that `80000064h` means communicating with the device in bus 0, device 0, function 0, and offset 64. This is the memory controller configuration register of the VIA 693A northbridge. In most circumstances, the PCI device that occupies bus 0, device 0, function 0 is the host bridge. However, you need to consult the chipset datasheet to verify this. The next routines are easy to understand. If you still feel confused, I suggest that you learn a bit more of x86 assembly language. In general, the code does the following: it reads the offset data, modifies it, and writes it back to the device.

The configuration space of every PCI device contains device-specific registers used to configure the device. Some registers within the 256-bytes configuration space possibly are not implemented and simply return `0xFF` on PCI configuration read cycles.

As you know, the amount of RAM can vary among systems. How can PCI devices handle this problem? How are they relocated to different addresses as needed? The answer lays in the PCI configuration space registers. Recall from figures 1.7 and 1.8 that the predefined configuration header contains a so-called BAR. These registers are responsible for PCI devices addressing. A BAR contains the starting address within the memory or I/O address space that will be used by the corresponding PCI device during its operation. The BAR contents can be read from and written into, i.e., they are programmable using software. It's the responsibility of the BIOS to initialize the BAR of every PCI device to the right value during boot time. The value must be unique and must not collide with the memory or I/O address that's used by another device or the RAM. Bit 0 in all BARs is read only and is used to determine whether the BARs map to the memory or I/O address space.

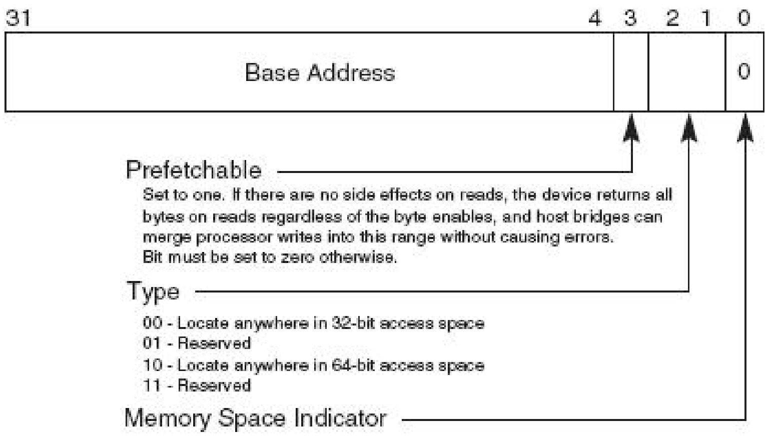

31
4 3 2 1 0
Base Address
0
Prefetchable
Set to one. If there are no side effects on reads, the device returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes into this range without causing errors. Bit must be set to zero otherwise.
Type
00 - Locate anywhere in 32-bit access space
01 - Reserved
10 - Locate anywhere in 64-bit access space
11 - Reserved
Memory Space Indicator

**Figure 1.11 Format of BAR that maps to memory space**

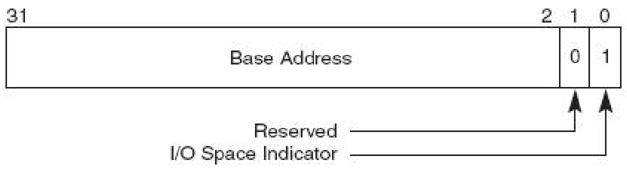


**Figure 1.12 Format of BAR that maps to I/O space**

Note that 64-bit PCI devices are implemented by using two consecutive BARs and can only map to the memory address space. A single PCI device can implement several BARs to be mapped to memory space while the remaining BAR is mapped to I/O space. This shows that the presence of BAR enables any PCI device to be relocatable within the system-wide memory and I/O address space.

How can BIOS initialize the address space usage of a single PCI device, since BAR only contains the lower limit of the address space that will be used by the device? How does the BIOS know how much address space will be needed by a PCI device? BAR contains *programmable bits* and *bits hardwired to zero*. The programmable bits are the most significant bits, and the hardwired bits are the least significant bits. The implementation note taken from PCI specification version 2.3 is as follows:

***Implementation Note: Sizing a 32-bit Base Address Register Example***

*Decode (I/O or memory) of a register is disabled via the command register before sizing a Base Address register. Software saves the original value of* *the Base Address register, writes 0FFFFFFFFh to the register, then reads it back. Size calculation can be done from the 32-bit value read by first clearing encoding information bits (bit 0 for I/O, bits 0–3 for memory), inverting all 32 bits (logical NOT), then incrementing by 1. The resultant 32-bit value is the memory–I/O range size decoded by the register. Note that the upper 16 bits of the result are ignored if the Base Address register is for I/O and bits 16–31 returned zero upon read. The original value in the Base Address register is restored before reenabling decode in the command register of the device.*

*64-bit (memory) Base Address registers can be handled the same, except that the second 32-bit register is considered an extension of the first; i.e., bits 32–63. Software writes 0FFFFFFFFh to both registers, reads them back, and combines the result into a 64-bit value. Size calculation is done on the 64-bit value.*

It's clear from the preceding implementation note that the BIOS can "interrogate" the PCI device to know the address space consumption of a PCI device. Upon knowing this information, BIOS can program the BAR to an unused address within the processor address space. Then, with the consumption information for the address space, the BIOS can program the next BAR to be placed in the next unused address space above the previous BAR address. The latter BAR must be located at least in the address that's calculated with the following formula:

```
next_BAR = previous_BAR + previous_BAR_address_space_consumption + 1
```

However, it's valid to program the BAR above the address calculated with the preceding formula. With these, the whole system address map will be functioning flawlessly. This relocatable element is one of the key properties that the PCI device brings to the table to eliminate the address space collision that once was the nightmare of ISA devices.

### 1.4.3. Proprietary Interchipset Protocol Technology

Motherboard chipset vendors have developed their own proprietary interchipset protocol between the northbridge and the southbridge in these last few years, such as VIA with V-Link, SiS with MuTIOL, and Intel with hub interface (HI). *These protocols are only an interim solution to the bandwidth problem between the peripherals that reside in the PCI expansion slots, on-board PCI chips, and the main memory, i.e., system RAM.* With the presence of newer and faster bus protocols such as PCI Express and HyperTransport in the market, these interim solutions are rapidly being phased out. However, reviewing them is important to clean up issues that might plague you once you discover the problem of understanding how it fits to the BIOS scene.

These proprietary protocols are transparent from configuration and initialization standpoints. They do not come up with something new. All are employing a PCI configuration mechanism to configure PCI compliant devices connected to the northbridge and southbridge. The interchipset link in most cases is viewed as a bus connecting the northbridge and the southbridge. This “protocol transparency” is needed to minimize the effect of the protocol on the investment needed to implement it. As an example, the Intel 865PE-ICH5 chipset mentioned this property clearly in the i865PE datasheet, as follows:

*In some previous chipsets, the "MCH" and the "I/O Controller Hub (ICHx)" were physically connected by PCI bus 0. From a configuration standpoint, both components appeared to be on PCI bus 0, which was also the system's primary PCI expansion bus. The MCH contained two PCI devices while the ICHx was considered one PCI device with multiple functions.*
*In the 865PE/865P chipset platform the configuration structure is significantly different. The MCH and the ICH5 are physically connected by the hub interface, so, from a configuration standpoint, the hub interface is logically PCI bus 0. As a result, all devices internal to the MCH and ICHx appear to be on PCI bus 0. The system's primary PCI expansion bus is physically attached to the ICH5 and, from a configuration perspective, appears to be a hierarchical PCI bus behind a PCI-to-PCI bridge; therefore, it has a programmable PCI Bus number. Note that the primary PCI bus is referred to as PCI_A in this document and is* ***not*** *PCI bus 0 from a configuration standpoint. The AGP [accelerated graphics port] appears to system software to be a real PCI bus behind PCI-to-PCI bridges resident as devices on PCI bus 0.*
*The MCH contains four PCI devices within a single physical component.*

Further information regarding these protocols can be found in the corresponding chipset datasheets. Perhaps, some chipset's datasheet does not mention this property clearly. Nevertheless, by analogy, you can conclude that those chipsets must have adhered to the same principle.

### 1.4.4. PCI Express Bus Protocol

The PCI Express protocol supports the PCI configuration mechanism explained in the previous subsection. Thus, in PCI Express–based systems, the PCI configuration mechanism is still used. In most cases, to enable the new PCI Express–enhanced configuration mechanism, the BIOS has to initialize some critical PCI Express registers by using the PCI configuration mechanism before proceeding to use the PCI Express–enhanced configuration mechanism. It's necessary because the new PCI Express–enhanced configuration mechanism uses BARs that have to be initialized to a known address in the system-wide address space before the new PCI Express–enhanced configuration cycle.

PCI Express devices, including PCI Express chipsets, use the so-called root complex register block (RCRB) for device configuration. The registers in the RCRB are *memory-mapped registers*. Contrary to the PCI configuration mechanism that uses I/O read/write transactions, the PCI Express–enhanced configuration mechanism uses memory read/write transactions to access any register in the RCRB. However, the read/write instructions must be carried out in a 32-bit boundary, i.e., must not cross the 32-bit natural boundary in the memory address space. A root complex base address register (RCBAR) is used to address the RCRB in the memory address space. The RCBAR is configured using the PCI configuration mechanism. Thus, the algorithm used to configure any register in the RCRB as follows:

1. Initialize the RCBAR in the PCI Express device to a known address in the memory address space by using the PCI configuration mechanism.
2. Perform a memory read or write on 32-bit boundary to the memory-mapped register by taking into account the RCBAR value; i.e., the address of the register in the memory address space is equal to the RCBAR value plus the offset of the register in the RCRB.

Perhaps, even the preceding algorithm is still confusing. Thus, a sample code is provided in listing 1.2.

**Listing 1.2 PCI Express–Enhanced Configuration Access Sample Code**

```
Init_HI_RTC_Regs_Mapping proc near
  mov   eax, 8000F8F0h        ; Enable the PCI configuration cycle to
                              ; bus 0, device 31, function 0, i.e.,
                              ; the LPC bridge in Intel ICH7
  mov   dx, 0CF8h             ; dx = PCI configuration address port
  out   dx, eax
  add   dx, 4                 ; dx = PCI configuration data port
  mov   eax, 0FED1C001h       ; enable root complex configuration
                              ; base address at memory space FED1_C000h
  out   dx, eax
  mov   di, offset ret_addr_1 ; Save return address to di register
  jmp   enter_flat_real_mode
; -------------------------------------------------------------------
ret_addr_1:
  mov   esi, 0FED1F400h       ; RTC configuration (ICH7 configuration
                              ; register at memory space offset 3400h)
  mov   eax, es:[esi]
  or    eax, 4                ; Enable access to upper 128 bytes of RTC
  mov   es:[esi], eax
  mov   di, offset ret_addr_2 ; Save return address to di register
  jmp   exit_flat_real_mode
; -------------------------------------------------------------------
ret_addr_2:
  mov   al, 0A1h
  out   72h, al
  out   0EBh, al
  in    al, 73h
  out   0EBh, al              ; Show the CMOS value in a diagnostic port
  mov   bh, al
  retn
Init_HI_RTC_Regs_Mapping endp
```

Listing 1.2 is a code snippet from a disassembled boot block part of the Foxconn 955X7AA-8EKRS2 motherboard BIOS. This motherboard is based on Intel 955X-ICH7 chipsets. As you can see, the register that controls the RTC register in the ICH7[7] is a memory-mapped register and accessed by using a memory read or write instruction as per the PCI Express–enhanced configuration mechanism. In the preceding code snippet, the ICH7 RCRB base address is initialized to `FED1_C000h`. *Note that the value of the last bit is an enable bit and not used in the base address calculation.* That's why it has to be set to one to enable the root-complex configuration cycle. This technique is analogous to the PCI configuration mechanism. The root-complex base address is located in the memory address space of the system from a CPU perspective.

One thing to note is that the PCI Express–enhanced configuration mechanism described here is implementation dependent; i.e., it works in the Intel 955X-ICH7 chipset. Future chipsets may implement it in a different fashion. Nevertheless, you can read the PCI Express specification to overcome that. Furthermore, another kind of PCI Express–enhanced configuration mechanism won't differ much from the current example. The registers will be memory mapped, and there will be an RCBAR.

### 1.4.5. HyperTransport Bus Protocol

In most cases, the HyperTransport configuration mechanism uses the PCI configuration mechanism that you learned about in the previous section. Even though the HyperTransport configuration mechanism is implemented as a memory-mapped transaction under the hood, it's transparent to programmers; i.e., there are no major differences between it and the PCI configuration mechanism. HyperTransport-specific configuration registers are also located in within the 256-byte PCI configuration registers. However, HyperTransport configuration registers are placed at higher indexes than those used for mandatory PCI header, i.e., placed above the first 16 dwords in the PCI configuration space of the corresponding device. These HyperTransport-specific configuration registers are implemented as new capabilities, i.e., pointed to by the capabilities pointer[8] in the device's PCI configuration space. Please refer to figure 1.7 for the complete PCI configuration register layout.

[7] The RTC control register is located in the LPC bridge. The LPC bridge in ICH7 is device 31, function 0.
[8] The capabilities pointer is located at offset 34h in the standard PCI configuration register layout.