System Address Map Initialization in x86/x64 Architecture Part 1: PCI-Based Systems

POSTED IN HACKING ON SEPTEMBER 16, 2013

SHARE

This article serves as a clarification about the PCI expansion ROM address mapping, which was not sufficiently covered in my "Malicious PCI Expansion ROM" article published by Infosec Institute last year (http://resources.infosecinstitute.com/pci-expansion-rom/). Low-level programmers are sometimes puzzled about the mapping of device memory, such as PCI device memory, to the system address map. This article explains the initialization of the system address map, focusing on the initialization of the PCI chip registers that control PCI device memory address mapping to the system address map. PCI device memory address mapping is only required if the PCI device contains memory, such as a video card, network card with onboard buffer, or network card that supports PCI expansion ROM, etc.

ETHICAL HACKING TRAINING – RESOURCES (INFOSEC)

X86/x64 system address map is complex due to backward compatibility that must be maintained in the bus protocol in x86/x64 architecture. Bus protocol being utilized in a system dictates the address mapping of the memory of a device—that's attached to the bus—to the system address map. Therefore, you must understand the address mapping mechanism of the specific bus protocol to understand the system address map initialization. *This article focuses on systems based on the PCI bus protocol*. PCI bus protocol is a legacy bus protocol by today's standard. However, it's very important to understand how it works in the lowest level in terms of software/firmware, because it's impossible to understand the later bus protocol, the PCI Express (PCIe) without understanding PCI bus protocol. PCIe is virtually the main bus protocol in every x86/x64 systems today. Part 2 of this article will focus on PCIe-based systems.

ETHICAL HACKING TRAINING – RESOURCES (INFOSEC)

Conventions

There are several different usages of the word “memory” in this article. It can be confusing for those new to the subject. Therefore, this article uses these conventions:

1. The word “main memory” refers to the RAM modules installed on the motherboard.
2. The word “memory controller” refers to part of the chipset or the CPU that controls the RAM modules and accesses to the RAM modules.
3. Flash memory refers to either the chip on the motherboard that stores the BIOS/UEFI or the chip that stores the PCI expansion ROM contents.
4. The word “memory range” or “memory address range” means the range, i.e., from the base/start address to the end address (base address + memory size) occupied by a device in the CPU memory space.
5. The word “memory space” means the set of memory addresses accessible by the CPU, i.e., the memory that is addressable from the CPU. Memory in

this context could mean RAM, ROM or other forms of memory which can be addressed by the CPU.

6. The word "PCI expansion ROM" mostly refers to the ROM chip on a PCI device, except when the context contains other specific explanation.

The Boot Process at a Glance

This section explains the boot process in sufficient detail to understand the system address map and other bus protocol-related matters that are explained later in this article. You need to have a clear understanding of the boot process before we get into the system address map and bus protocol-related talks.

The boot process in x86/x64 starts with the platform firmware (BIOS/UEFI) execution. The platform firmware execution happens prior to the operating system (OS) boot, specifically before the "boot loader" loads and executes the OS. Platform firmware execution can be summarized as follows:

1. Start of execution in the CPU (processor) reset vector. In all platforms, the bootstrap processor (BSP) starts execution by fetching the instruction located in an address known as the *reset vector*. In x86/x64 this address is 4GB minus 16-bytes (`FFFF_FFF0h`). This address is always located in the BIOS/UEFI flash memory on the motherboard.
2. CPU operating mode initialization. In this stage, the platform firmware switches the CPU to the platform firmware CPU operating mode; it could be real mode, "voodoo" mode, or flat protected mode, depending on the platform firmware. X86/x64 CPU resets in a modified real mode operating mode, i.e., real mode at physical address `FFFF_FFF0h`. Therefore, if the platform firmware CPU operating mode is flat protected mode, it must switch the CPU into that mode. Present-day platform firmware doesn't use "voodoo" mode as extensively as in the past. In fact, most present-day platform firmware has abandoned its use altogether. For example, UEFI implementations use flat protected mode.
3. Preparation for memory initialization. In this stage there are usually three steps carried out by the platform firmware code:
   A. CPU microcode update. In this step the platform firmware loads the CPU microcode update to the CPU.
   B. CPU-specific initialization. In x86/x64 CPUs since (at least) the Pentium III and AMD Athlon era, part of the code in this stage usually sets up a temporary stack known as cache-as-RAM (CAR), i.e.,

the CPU cache acts as temporary (writeable) RAM because at this point of execution there is no writable memory—the RAM hasn't been initialized yet. Complex code in the platform firmware requires the use of a stack. In old BIOS, there is some sort of assembler macro trick for return address handling because by default the return address from a function call in x86/x64 is stored in a "read only" stack, but no writeable memory variable can be used. However, this old trick is not needed anymore, because all present-day CPUs support CAR. If you want to know more about CAR, you can consult the BIOS and Kernel Developer Guide (BKDG) for AMD Family 10h over at http://support.amd.com/us/Processor_TechDocs/31116.pdf. Section 2.3.3 of that document explains how to use the CPU L2 cache as general storage on boot. CAR is required because main memory (RAM) initialization is a complex task and requires the use of complex code as well. The presence of CAR is an invaluable help here. Aside from CAR setup, certain CPUs need to initialize some of its machine-specific registers (MSRs); the initialization is usually carried out in this step.

C. Chipset initialization. In this step the chipset registers are initialized, particularly the chipset base address register (BAR). We'll have a look deeper into BAR later. For the time being, it's sufficient that you know BAR controls how the chip registers and memory (if the device has its own memory) are mapped to the system address map. In some chipsets, there is a watch dog timer that must be disabled before memory initialization because it could randomly reset the system. In that case, disabling the watch dog timer is carried out in this step.

4. Main memory (RAM) initialization. In this step, the memory controller initialization happens. In the past, the memory controller was part of the chipset. Today, that's no longer the case. The memory controller today is integrated into the CPU. The memory controller initialization and RAM initialization happens together as complementary code, because the platform firmware code must figure out the correct parameters supported by both the memory controller and the RAM modules installed on the system and then initialize both of the components into the "correct" setup.
5. Post memory initialization. Before this step, the platform firmware code is executed from the flash ROM in the motherboard—and if CAR is enabled, the CPU cache acts as the stack. That's painfully slow compared to

"ordinary" code execution in RAM, especially with instructions fetched into the CPU, because the flash ROM is very slow compared to RAM. Therefore, the platform firmware binary usually copies itself to RAM in this step and continues execution there. In the previous step, the main memory (RAM) is initialized. However, there are several more steps required before the main memory (RAM) can be used to execute the platform firmware code:

A. Memory test. This is a test performed to make sure RAM is ready to be used because it's possible that some parts of the RAM are broken. The detail of how the test is carried out depends on the boot time requirement of the system. If the boot time requirement is very fast, in many cases it's impossible to test all parts of the RAM and only some parts can be tested with some sort of statistical approach on which parts to test to make sure the test covers as wide parts as possible (statistically speaking).
B. "Shadowing" the firmware to RAM. "Shadowing" in this context means copying the RAM from the flash ROM to the RAM at address range below the 1MB limit—1 mb is the old 20-bit address mapping limit set for DOS-era hardware. However, the copying is *not a trivial copy*, because the code will reside in the RAM but in the same address range previously occupied by the flash ROM—this is why it's called "shadowing." Some bit twiddling must be done on the chipset by the platform firmware code to control the mapping of the address range to the RAM and the flash ROM. Details of the "bit twiddling" are outside the scope of this article. You can read details of the mapping in the respective chipset datasheet.
C. Redirecting memory transaction to the correct target. This is a continuation of the "shadowing" step. The details depends on the platform (CPU and chipset combination), and the runtime setup, i.e., whether to shadow the platform firmware or not at runtime (when the OS runs).
D. Setting up the stack. This step sets up the stack (in RAM) to be used for further platform firmware code execution. In previous steps, the stack is assumed to be present in the CAR. In this step, the stack is switched from CAR to RAM because the RAM is ready to be used. This is important because the space for stack in CAR is limited compared to RAM.
E. Transferring platform firmware execution to RAM. This is a "jump" to the platform firmware code which is "shadowed" to the RAM in step b.

6. Miscellaneous platform enabling. This step depends on the specific system configuration, i.e., the motherboard and supporting chips. Usually, it consists of clock generator chip initialization, to run the platform at the intended speed, and in some platforms this step also consists of initializing the general purpose I/O (GPIO) registers.
7. Interrupt enabling. Previous steps assume that the interrupt is not yet enabled because all of the interrupt hardware is not yet configured. In this step the interrupt hardware such as the interrupt controller(s) and the associated interrupt handler software are initialized. There are several possible interrupt controller hardware in x86/x64, i.e., the 8259 programmable interrupt controller (PIC), the local advanced programmable interrupt controller (LAPIC) present in most CPUs today, and the I/O advanced programmable interrupt controller (IOxAPIC) present in most chipsets today. After the hardware and software required to handle the interrupt are ready, the interrupt is enabled.
8. Timer initialization. In this step, the hardware timer is enabled. The timer generates timer interrupt when certain interval is reached. OS and some applications running on top of the OS use the timer to work. There are also several possible pieces of hardware (or some combination) that could act as the timer in x86/x64 platform, i.e., the 8254 programmable interrupt timer (PIT) chip that resides in the chipset, the high precision event timer (HPET) also residing in the chipset—this timer doesn't need initialization and is used only by the OS, real time clock (RTC) which also resides in the chipset, and the local APIC (LAPIC) timer present in the CPU.
9. Memory caching control initialization. X86/x64 CPU contains memory type range registers (MTRRs) that controls the caching of all memory ranges addressable by the CPU. The caching of the memory ranges depends on the type of hardware present in the respective memory range and it must be initialized accordingly. For example, the memory range(s) occupied by I/O devices such as the PCI bus must be initialized as uncached address range(s)—memory range(s) in this context is as seen from the CPU point of view.
10. Application processor(s) initialization. The non-bootstrap CPU (processor) core is called the application processor (AP) in some documentation; we will use the same naming here. In multicore x86/x64 CPUs, only the BSP is active upon reset. Therefore, the other cores—the AP—must be initialized accordingly before the OS boot-loader takes control of the system. One of the most important things to initialize in the AP is the MTRRs. The MTRRs must be consistent in all CPU cores, otherwise memory read and write

could misbehave and bring the system to a halt.

11. "Simple" I/O devices initialization. "Simple" IO devices in this context are hardware such as super IO (SIO), embedded controller, etc. This initialization depends on the system configuration. The SIO typically controls legacy IO, such as PS/2 and serial interfaces, etc. The embedded controller is mostly found on laptops, it controls things such as buttons on the laptop, the interface from the laptop motherboard to the battery, etc.
12. PCI device discovery and initialization. In this step, PCI devices—by extension the PCIe devices and other devices connected to PCI-compatible bus—are detected and initialized. The devices detected in this step could be part of the chipset and/or other PCI devices in the system, either soldered to the motherboard or in the PCI/PCIe expansion slots. There are several resource assignments to the device happening in this step: IO space assignment, memory mapped IO (MMIO) space assignment, IRQ assignment (for devices that requires IRQ), and expansion ROM detection and execution. The assignment of memory or IO address space happens via the use of BAR. We'll get into the detail in the PCI bus base address registers initialization section. USB devices initialization happens in this step as well because USB is a PCI bus-compatible protocol. Other non-legacy devices are initialized in this step as well, such as SATA, SPI, etc.
13. OS boot-loader execution. This is where the platform firmware hands over the execution to the OS boot-loader, such as GRUB or LILO in Linux or the Windows OS loader.

Now, the boot process carried out by the platform firmware should be clear to you. Particularly the steps where the system address map is initialized in relation to PCI devices, namely step 3c and step 12. All of these steps deal with the BAR in the PCI chip or part of the chipset.

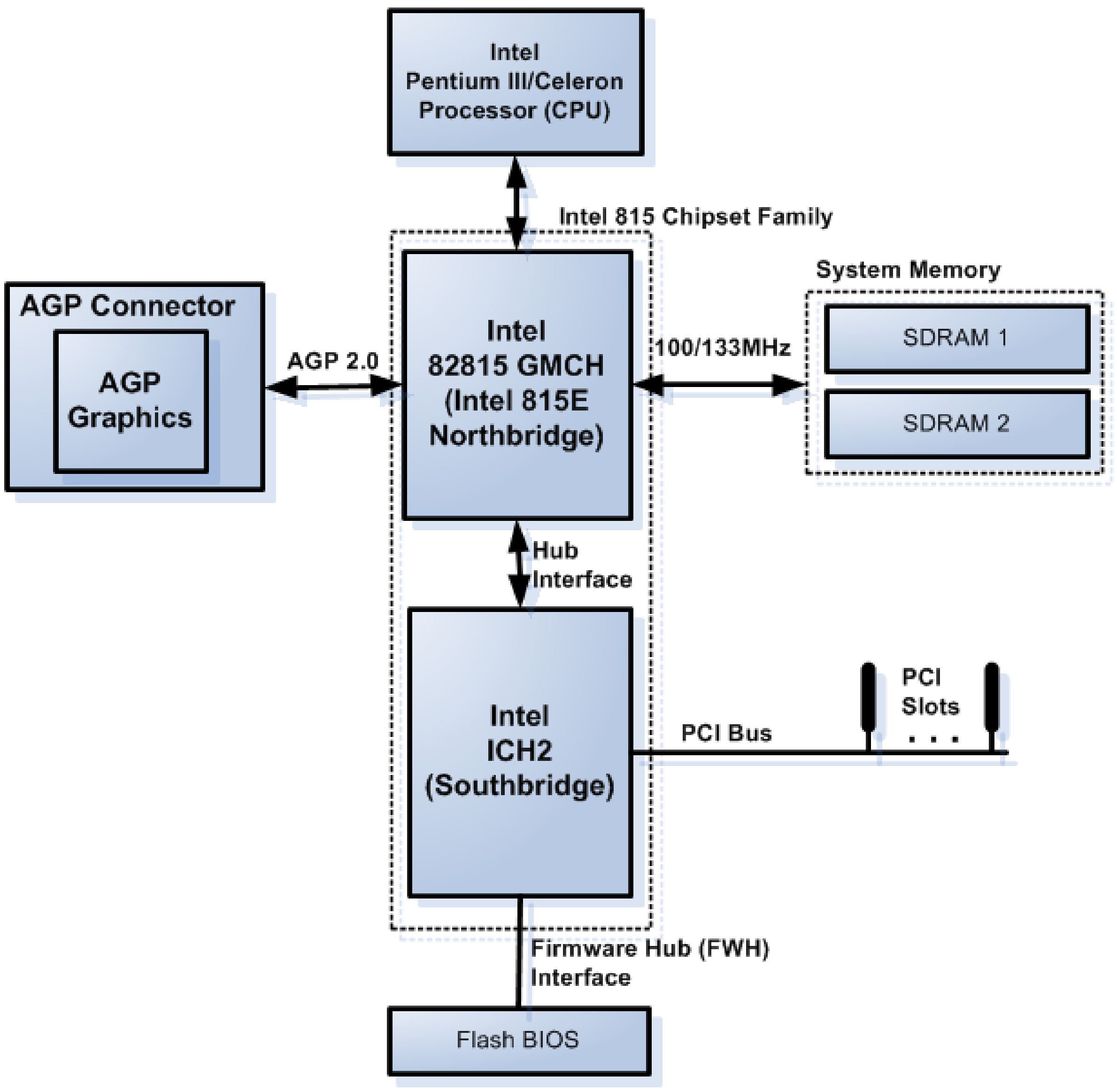
Dissecting PCI-Based System Address Map

We need to dissect a typical implementation sample of the system address map in x86/x64 before we can proceed to the system address map initialization in more detail. The Intel 815E chipset is the implementation sample here. You can download the chipset datasheet at http://download.intel.com/design/chipsets/datashts/29068801.pdf. Intel 815E is only half of the equation because it's the "northbridge" part of the chipset. The "southbridge" part of the chipset is Intel ICH2. You can download Intel ICH2 datasheet at http://www.intel.com/content/www/us/en/chipsets/82801ba-i-o-controller-hub-2-82801bam-i-o-controller-hub-2-mobile-datasheet.html. The northbridge is the chipset closer to the CPU and connected directly to the CPU, while the southbridge is the chipset farther away from the CPU and connected to the CPU via the northbridge. In present-day CPUs, the northbridge is typically integrated into the CPU package.

System Based on Intel 815E-ICH2 Chipset

Figure 1 shows the "simplified" block diagram of the system that uses Intel 815E-ICH2 chipset combination. Figure 1 doesn't show the entire connection from the chipset to other components in the system, *only* those related to the address mapping in the system.

Figure 1 Intel 815E-ICH2 (Simplified) Block Diagram



The Intel 815E-ICH2 chipset pair is not a "pure" PCI chipset, because it implements a non-PCI bus to connect the northbridge and the southbridge, called the hub interface (HI), as you can see in Figure 1. However, from logic point of view, the HI bus is basically a PCI bus with faster transfer speed. Well, our focus here is not on the transfer speed or techniques related to data transfers, but on the address mapping and, since HI doesn't alter anything related to address mapping, we can safely ignore the HI bus specifics and regard it in the same respect as PCI bus.

The Intel 815E chipset is ancient by present standards, but it's very interesting case study for those new to PCI chipset low-level details because it's very close to "pure" PCI-based systems. As you can see in Figure 1, Intel 815E chipset was one of the northbridge chipset for Intel CPUs that uses socket 370, such as Pentium III (code name Coppermine) and Intel Celeron CPUs.

Figure 1 shows how the CPU connects (logically) to the rest of the system via the Intel 815E northbridge. This implies that any access to any device outside the CPU must pass through the northbridge. From an address mapping standpoint, this means that Intel 815E acts as a sort of "address mapping router," i.e., the device that routes read or write transactions to a certain address—or address range(s)—to the correct device. In fact, that is how the northbridge works in practice. The difference with present-day systems is the physical location of the northbridge, which is integrated into the CPU, instead of being an individual component on the motherboard like Intel 815E back then. The configuration of the "address mapping router" in the northbridge at boot differs from the runtime configuration. In practice, the "address mapping router" is a series of (logical) PCI device registers in the northbridge that control the system address map. Usually, these registers are part of the memory controller and the AGP/PCI Bridge logical devices in the northbridge chipset. The platform firmware initializes these address mapping-related registers at boot to prepare for runtime usage inside an OS.

Now that you know how the system address map works at the physical level, i.e., it is controlled by registers in the northbridge, it's time to dive into the system address map itself. Figure 2 shows the system address map of systems using Intel 815E chipset.

Figure 2 Intel 815E System Address Map

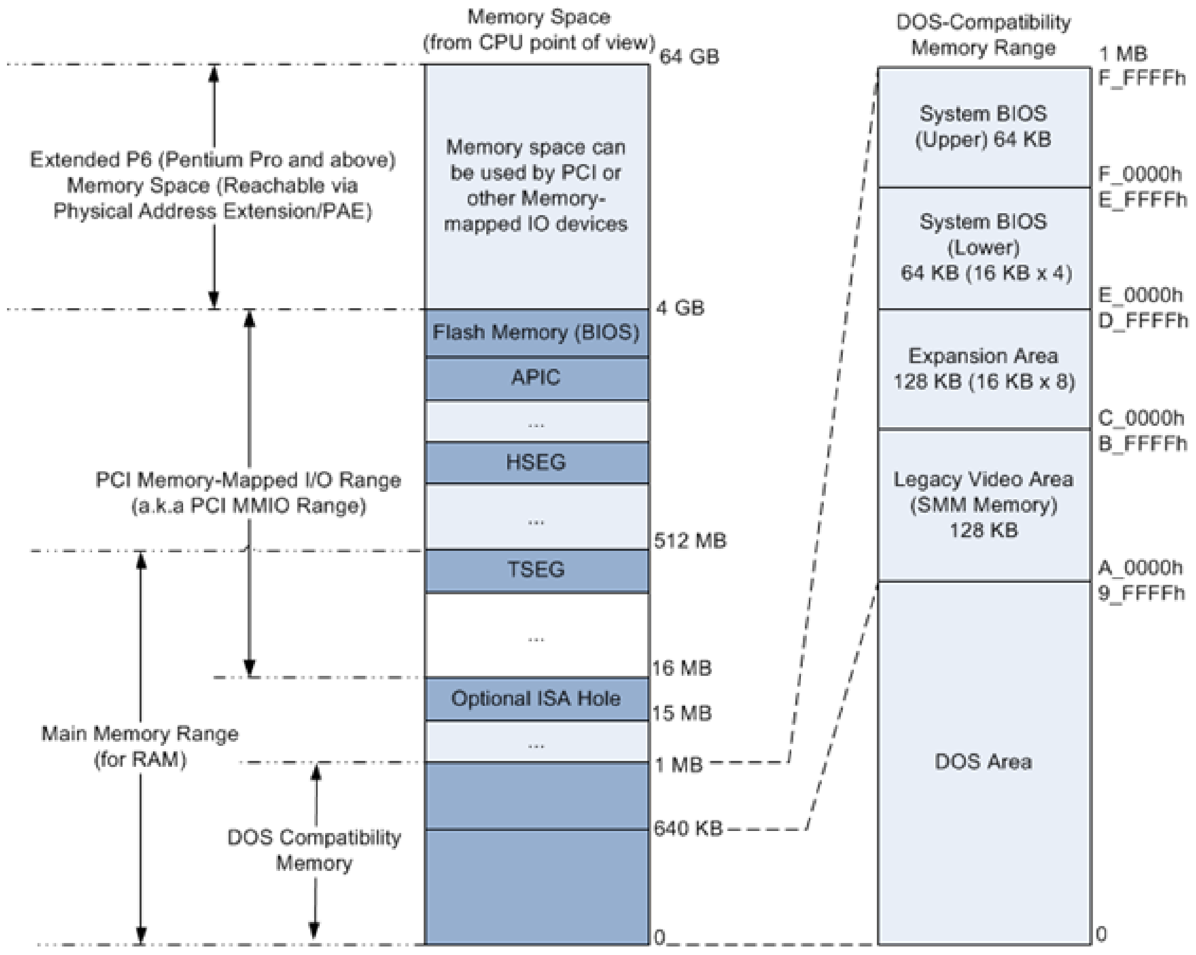

Memory Space (from CPU point of view)
64 GB
Extended P6 (Pentium Pro and above) Memory Space (Reachable via Physical Address Extension/PAE)
Memory space can be used by PCI or other Memory-mapped IO devices
4 GB
Flash Memory (BIOS)
APIC
HSEG
PCI Memory-Mapped I/O Range (a.k.a PCI MMIO Range)
512 MB
TSEG
16 MB
Optional ISA Hole
15 MB
Main Memory Range (for RAM)
1 MB
640 KB
DOS Compatibility Memory
0
DOS-Compatibility Memory Range
1 MB F_FFFFh
System BIOS (Upper) 64 KB
F_0000h E_FFFFh
System BIOS (Lower) 64 KB (16 KB x 4)
E_0000h D_FFFFh
Expansion Area 128 KB (16 KB x 8)
C_0000h B_FFFFh
Legacy Video Area (SMM Memory) 128 KB
A_0000h 9_FFFFh
DOS Area
0

Figure 2 shows Intel 815E system address map. You can find a complementary address mapping explanation in Intel 815E chipset datasheet in Section 4.1, “System Address Map.” Anyway, you should be aware that the system address map is “seen” from the CPU point of view, not from other chip in the system.

Figure 2 shows that the memory space supported by the CPU is actually 64GB. This support has been in Intel CPUs since the Pentium Pro era, by using a technique called physical address extension (PAE). Despite that, the amount of memory space used depends on the memory controller in the system, which in this case located in the northbridge (Intel 815E chipset). The Intel 815E chipset only supports up to 512MB of main memory (RAM) and only uses a 4GB memory space. Figure 2 also shows that PCI devices consume (use) the CPU memory space. A device that consumes CPU memory space is termed a memory mapped I/O device or MMIO device for short. The MMIO term is widely used in the industry and applies to all other CPUs, not just x86/x64.

Figure 2 shows that the RAM occupies (at most) the lowest 512MB of the memory space. However, above the 16MB physical address, the space *seems to be shared* between RAM and PCI devices. Actually, there is *no sharing of memory range happening in the system* because the platform firmware initializes the PCI device’s memory to use memory range *above* the memory range consumed by main memory (RAM) in the CPU memory space. This memory range "free" from RAM depends on the amount of RAM installed in the system. If the installed RAM size is 256MB, the PCI memory range starts right after the 256MB boundary up until the 4GB memory space boundary, if the installed RAM size is 384MB, the PCI memory range starts right after the 384MB boundary up until the 4GB memory space boundary and so on. Therefore, this implies that the memory range consumed by the PCI devices is relocatable, i.e., can be relocated within the CPU memory space, otherwise it's impossible to prevent conflict in memory range usage. Well, it's true and actually it's one of the features of the PCI bus which sets it apart from the ISA bus that it replaced. In the—very old—ISA bus, you have to set the jumpers on the ISA device to the correct setting; otherwise there will be address usage conflict in your system. In PCI bus, it's the job of the platform firmware to set up the correct address mapping in the PCI devices.

There are several special ranges in the memory range consumed by PCI device memory above the installed RAM size—installed RAM size is termed top of main memory (TOM) in Intel documentation, so I'll use the same term from now on. Some of the special ranges above TOM are hardcoded and cannot be changed because the CPU reset vector and certain non-CPU chip registers *always* map to those special memory ranges, i.e., they are all not relocatable. Among these "hardcoded" memory ranges are the memory ranges used by advanced programmable interrupt controller (APIC) and the flash memory which stores the BIOS/UEFI code. These hardcoded memory ranges *cannot be used by PCI devices* at all.

PCI Configuration Register

Now, we have arrived in the core of our discussion: how does the PCI bus protocol map PCI devices memory to the system address map? The mapping is accomplished by using a set of PCI device registers called BAR (base address register). We will get into details of the BAR initialization in the "PCI Bus Base Address Registers Initialization" section later. Right now, we will look at details of the BAR implementation in PCI devices.

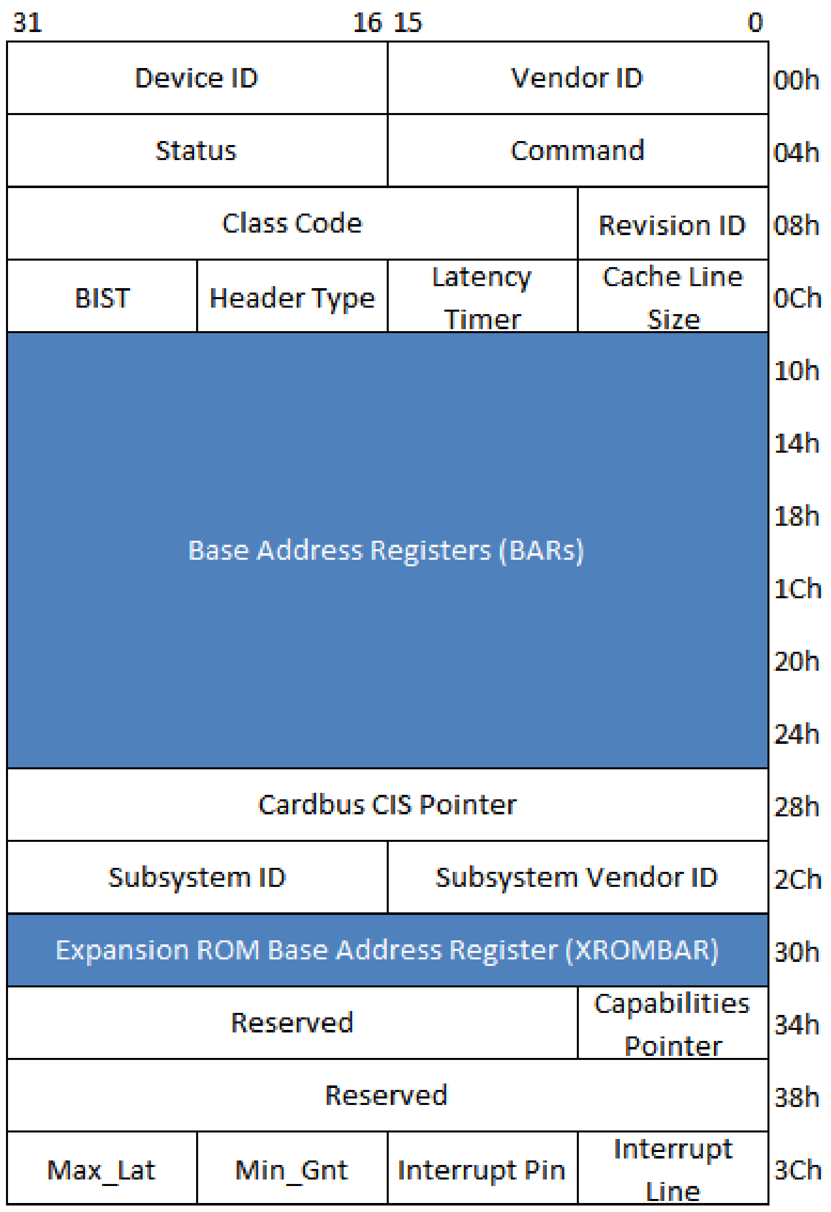
BARs are part of the so-called *PCI configuration register*. Every PCI device must implement the PCI configuration register dictated by the PCI specification. Otherwise, the device will not be regarded as a valid PCI device. The PCI configuration register controls the behavior of the PCI device at all times.

Therefore, changing the (R/W) value in the PCI configuration register would change the behavior of the system as well. In x86/x64 architecture, the PCI configuration register is accessible via two 32-bit I/O ports. I/O port CF8h-CFBh acts as address port, while I/O port CFCh-CFFh acts as the data port to read/write values into the PCI configuration register. For details on accessing the PCI configuration register in x86, please read this section of my past article: https://sites.google.com/site/pinczakko/pinczakko-s-guide-to-award-bios-reverse-engineering#PCI_BUS. The material in that link should give you a clearer view about access to the PCI configuration register in x86/x64 CPUs. Mind you that the code in that link must be executed in ring 0 (kernel mode) or under DOS (if you're using 16-bit assembler), otherwise it would make the OS respond with access permission exception.

Now, let's look more closely at the PCI configuration register. The PCI configuration register consists of 256 bytes of registers, from (byte) offset 00h to (byte) offset FFh. The 256-byte PCI configuration register consists of two parts, the first 64 bytes are called *PCI configuration register header* and the rest are called *device-specific PCI configuration register*. This article only deals with the BARs, which are located in the PCI configuration register header. It doesn't deal with the device-specific PCI configuration registers because only BARs affect the PCI device memory mapping to system address map.

There are two types of PCI configuration register header, a type 0 and a type 1 header. The PCI-to-PCI bridge device must implement the PCI configuration register type 1 header, while other PCI device must implement the PCI configuration register type 0 header. This article only deals with PCI configuration register type 0 header and focuses on the BARs. Figure 3 shows the PCI configuration register type 0 header. The registers are accessed via I/O ports CF8h-CFBh and CFCh-CFFh, as explained previously.

Figure 3 PCI Configuration Registers Type 0



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 (BARs)				10h
				14h
				18h
				1Ch
				20h
				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address Register (XROMBAR)				30h
Reserved			Capabilities Pointer	34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

Figure 3 shows that there are two types of BAR, highlighted on a blue background: the BARs themselves and the expansion ROM base address register (XROMBAR). BARs span the range of six 32-bit registers (24-bytes), from offset 10h to offset 27h in the PCI configuration header type 0. BARs are used for mapping the *non-expansion ROM PCI device memory*—usually RAM on the PCI device—to the system memory map, while XROMBAR is specifically used for mapping the PCI expansion ROM to system address map. It's the job of platform firmware to initialize the values of the BARs. Each BAR is a 32-bit registers, hence each of them can map PCI device memory in the 32-bit system address map, i.e., can map the PCI device memory to the 4GB memory address space.

The BARs and XROMBAR are readable and writeable registers, i.e., the contents of the BARs/XROMBAR can be modified. That's the core capability required to relocate the PCI device memory in the system address map. PCI device memory is said to be relocatable in the system address map, because you can change the “base” address (start address) of the PCI device memory in the system address map by changing the contents of the BARs/XROMBAR. It works like this:

- Different systems can have different main memory (RAM) size.
- Different RAM size means that the area in the system address map set aside for PCI MMIO range differs.
- Different PCI MMIO range means that the PCI device memory occupies different address in the CPU memory space. That means you have to be able to change the base address of the PCI device memory in the CPU memory space required when migrating the PCI device to a system with a different amount of RAM; the same is true if you add more RAM to the same system.
- BARs and XROMBAR control the address occupied by the PCI device memory. Because both of them are modifiable, you can change the memory range occupied by the PCI device memory (in the CPU memory space) as required.

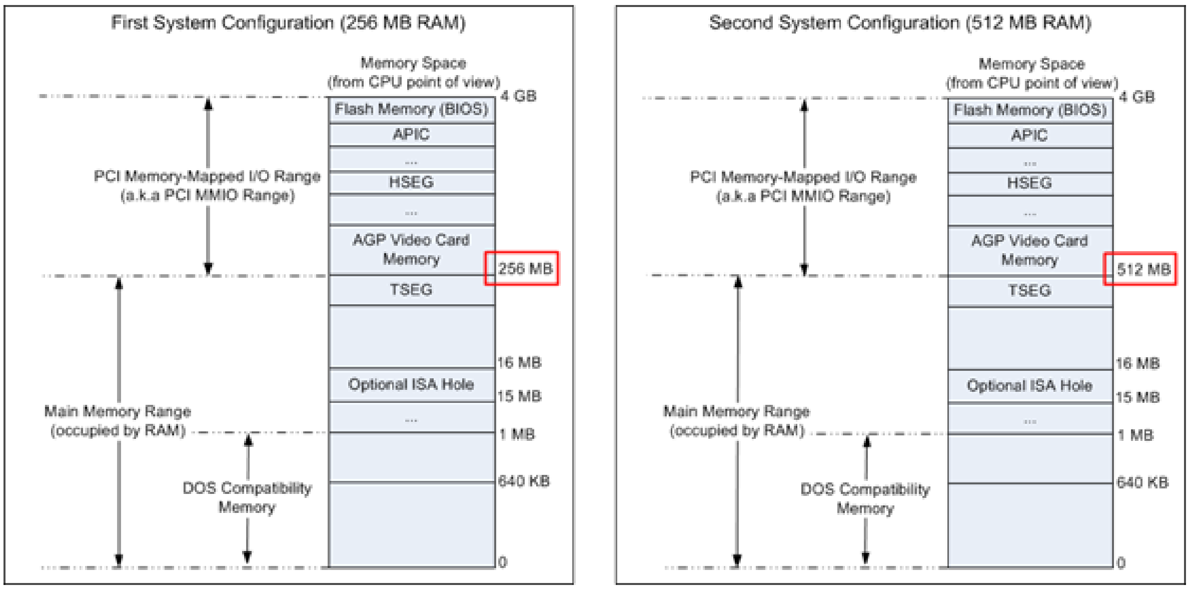
Practical System Address Map on Intel 815E-ICH2 Platform

Perhaps the XROMBAR and BARs explanation is still a bit confusing for beginners. Let’s look at some practical examples. The scenario is as follows:

1. The system in focus uses an Intel Pentium III CPU with 256 mb RAM, a motherboard with Intel 815E chipset, and an AGP video card with 32 mb onboard memory. The AGP video card is basically a PCI device with onboard memory from the system address map point of view. We call this configuration *the first system configuration* from now on.
2. The same system as in point 1. However, we add new 256 mb RAM module. Therefore, the system now has 512 mb RAM—the maximum amount of RAM supported by the Intel 815E chipset based on its datasheet. We call this configuration *the second system configuration* from now on.

We’ll have a look at what the system address map looks like in both of the configurations above. Figure 4 shows the system address map for the *first* system configuration (256 mb RAM) and the system address map for the *second* system configuration (512 mb RAM). As you can see, the memory range occupied by the PCI devices shrinks from 3840 mb (4GB – 256 mb) in the first system configuration (256 mb RAM) to 3584 mb (4GB – 512MB) in the second system configuration (512 mb RAM). The change also causes the base address of the AGP video card memory to change; in the first system configuration the base address is 256 mb while in the second system configuration the base address is 512 mb. The change in the AGP video card memory base address is possible because the contents of the AGP video card—its video chip—BAR can be modified.

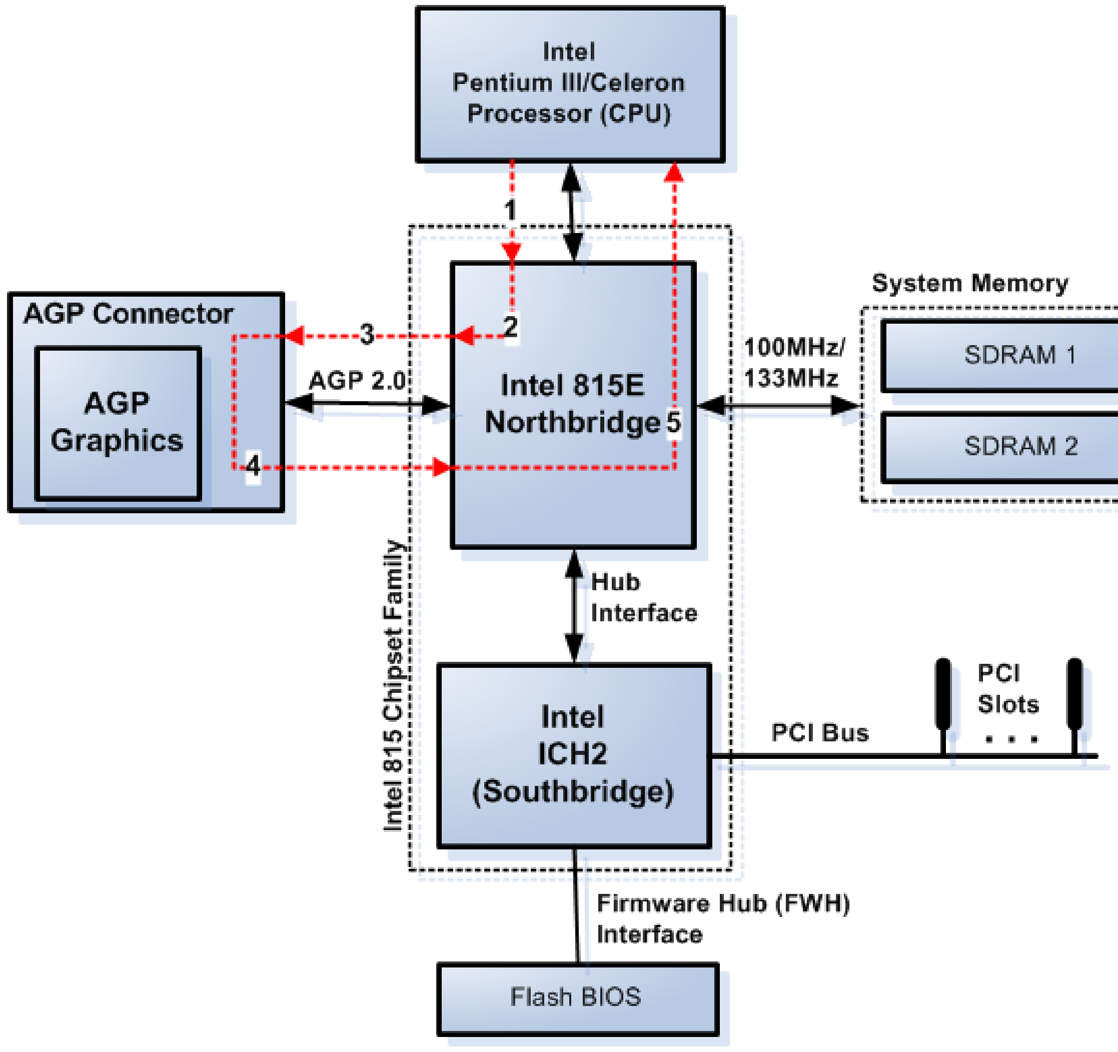
Figure 4 System Address Map for the First (256 mb RAM) and Second (512 mb RAM) System Configuration



First System Configuration (256 MB RAM)
Memory Space (from CPU point of view)
4 GB
Flash Memory (BIOS)
APIC
...
HSEG
...
AGP Video Card Memory
256 MB
TSEG
16 MB
Optional ISA Hole
15 MB
...
1 MB
640 KB
0
PCI Memory-Mapped I/O Range (a.k.a PCI MMIO Range)
Main Memory Range (occupied by RAM)
DOS Compatibility Memory
Second System Configuration (512 MB RAM)
Memory Space (from CPU point of view)
4 GB
Flash Memory (BIOS)
APIC
...
HSEG
...
AGP Video Card Memory
512 MB
TSEG
16 MB
Optional ISA Hole
15 MB
...
1 MB
640 KB
0
PCI Memory-Mapped I/O Range (a.k.a PCI MMIO Range)
Main Memory Range (occupied by RAM)
DOS Compatibility Memory

Now, let's see how the Intel 815E northbridge routes access to the CPU memory space in the first system configuration shown in Figure 4 (256 mb RAM). Let's breakdown the steps for a read request to the video (AGP) memory in the first system configuration. In the first system configuration, the platform firmware initializes the video memory to be mapped in memory range 256 mb to 288 mb, because the video memory size is 32 mb—the first 256 mb is mapped to RAM. The platform firmware does so by configuring/initializing the video chip BARs to accept accesses in the 256 mb to 288 mb memory range. Figure 5 shows the steps to read the contents of the video card memory starting at physical address `11C0_0000h` (284MB) at runtime (inside an OS).

Figure 5 Steps for a Memory Request to the AGP Video Card in the First System Configuration (256 mb RAM)



Details of the steps shown in Figure 5 are as follows:

1. Suppose that an application (running inside the OS) requests texture data in the video memory that ultimately translates into physical address `11C0_0000h` (284MB), the read transaction would go from the CPU to the northbridge.
2. The northbridge forwards the read request to a device whose memory range contains the requested address, `11C0_0000h`. The northbridge logic checks all of its registers related to address mapping to find the device that match the address requested. In the Intel 815E chipset, address mapping of device on the AGP port is controlled by four registers, in the AGP/PCI bridge part of the chipset. Those four registers are MBASE, MLIMIT, PMBASE, and PMLIMIT. Any access to a memory address within the range of either MBASE to MLIMIT or PMBASE to PMLIMIT is forwarded to the AGP. It turns out the requested address (`11C0_0000h`) is within the PMBASE to PMLIMIT memory range. Note that initialization of the four address mapping registers is the job of the platform firmware.
3. Because the requested address (`11C0_0000h`) is within the PMBASE to

PMLIMIT memory range, the northbridge forwards the read request to the AGP, i.e., to the video card chip.

4. The video card chip's BARs setting allows it to respond to the request from the northbridge. The video card chip then reads the video memory at the requested address (at `11C0_0000h`) and returns the requested contents to the northbridge.
5. The northbridge forwards the result returned by the video card to the CPU. After this step, the CPU receives the result from the northbridge and the read transaction completes.

The breakdown of a memory read in the sample above should be clear enough for those not yet accustomed to the role of the Intel 815E northbridge chipset as "address router."

The System Management Mode (SMM) Memory Mapping

In the beginning of this section, I have talked about the "special" memory range in the PCI memory range, i.e., the memory range occupied by the flash ROM chip containing the platform firmware and the APIC. If you look at the system address map in Figure 2, you can see that there are two more memory ranges in the system address map that show up mysteriously. They are the HSEG and TSEG memory ranges. Both of these memory ranges are not accessible in normal operating mode, i.e., not accessible from inside the operating system, even DOS. They are only accessible when the CPU is in system management mode (SMM). Therefore, it's only directly accessible to SMM code in the BIOS. Let's have a look at both of them:

1. HSEG is an abbreviation of high segment. This memory range is hardcoded to `FEEA_0000h`–`FEEB_FFFFh`. The *system management RAM control register* in the Intel 815E chipset must be programmed (at boot in the BIOS code) to enable this hardcoded memory range as HSEG, otherwise the memory range is mapped to the PCI bus. HSEG maps access to the `FEEA_0000h`–`FEEB_FFFFh` memory range into the `A_0000h`–`B_FFFFh` memory range in the RAM. By default, the `A_0000h`–`B_0000h` memory range is not mapped to RAM, but to part of the video memory to provide compatibility to DOS application. The HSEG capability makes the RAM "hole" in the `A_0000h`–`B_FFFFh` memory range available to store SMM

code. Both of the HSEG memory ranges are not accessible at runtime (inside an OS).

2. TSEG is an abbreviation of top of main memory segment—top of main memory (RAM) is abbreviated as TOM. This memory range is relocatable, depending on the size of main memory. It's always mapped from TOM-minus-TSEG-size to TOM. The *system management RAM control register* in the Intel 815E chipset must be programmed (at boot in the BIOS code) to enable TSEG. TSEG is only accessible in SMM with physical address set in the northbridge, the areas occupied by TSEG is a "memory hole" with respect to OS, i.e., OS cannot see TSEG area.

Accesses to all of the "special" memory ranges (APIC, flash ROM and HSEG) are not forwarded to the PCI bus by the Intel 815E-ICH2 chipset, but they are forwarded to their respective devices, i.e., accesses to APIC memory range are forwarded to the APIC, accesses to flash ROM memory range are forwarded to the flash ROM, and accesses to HSEG at `FEEA_0000h`–`FEEB_FFFFh` memory range are forwarded to RAM at `A_0000h`–`B_FFFFh` memory range.

At this point, everything regarding system address map in a typical Intel 815E-ICH2 system should be clear. The one thing remaining to be studied is initialization of the BAR. There are some additional materials in the remaining sections, though. They are all related to system address map.

PCI Bus Base Address Registers Initialization

PCI BAR initialization is the job of the platform firmware. The PCI specification provides implementation note on BAR initialization. Let's have a look at the BAR format before we get into the implementation note.

PCI BAR Formats

There are two types of BAR, the first is BAR that maps to the CPU I/O space and the second is BAR that maps to the CPU memory space. The formats of these two types of BARs are quite different. Figure 6 and Figure 7 show the formats of both types of BAR.

Figure 6 BAR That Maps to CPU I/O Space Format

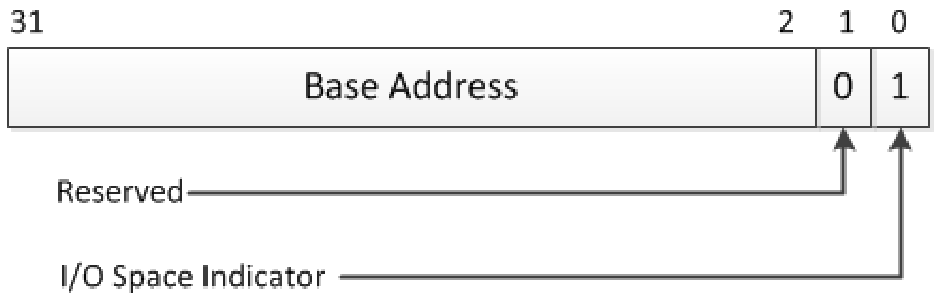


Figure 6 shows the format of the PCI BAR that maps to CPU I/O space. As you can see, the lowest two bits in the 32-bit BAR are hardcoded to "01" binary value. Actually, only the lowest bit matters; it differentiates the type of the BAR, because the bit has a different hardcoded value when the BAR types differ. In BAR that maps to CPU I/O space, the lowest bit always hardcoded to *one*, while in BAR that maps to CPU memory space, the lowest bit always hardcoded to *zero*. You can see this difference in Figure 6 and Figure 7.

Figure 7 BAR That Maps to CPU Memory Space Format

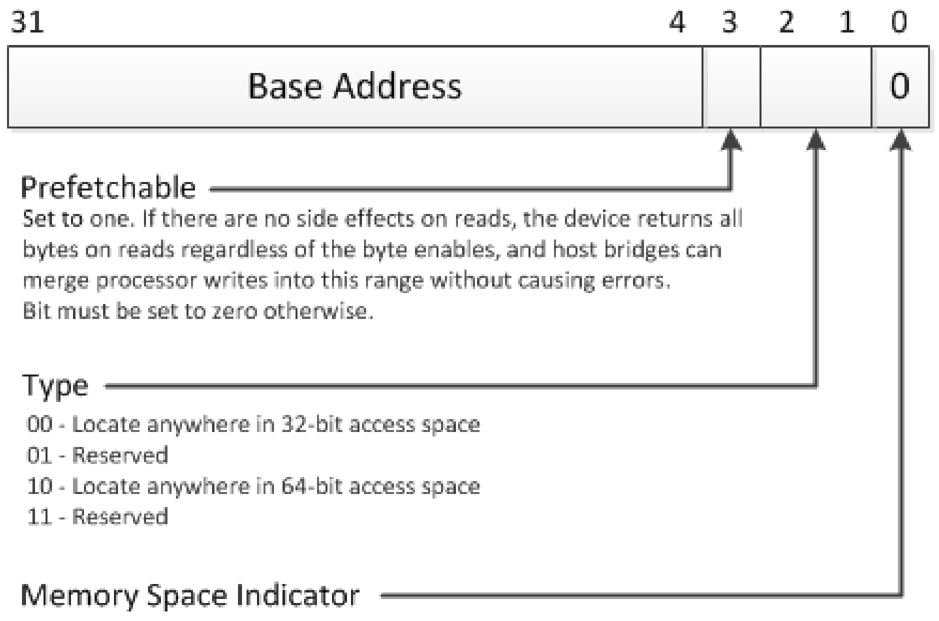

31
4
3
2
1
0
Base Address
0
Prefetchable
Set to one. If there are no side effects on reads, the device returns all bytes on reads regardless of the byte enables, and host bridges can merge processor writes into this range without causing errors.
Bit must be set to zero otherwise.
Type
00 - Locate anywhere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

Figure 7 shows the BAR format for the BAR that maps to CPU memory space. This article deals with this type of BAR because the focus is on the system address map, particularly the system memory map. Figure 7 shows the lowest bit is hardcoded to zero in BAR that map to CPU memory space. Figure 7 also shows that bits 1 and bit 2 determine whether the BAR is a 32-bit BAR or 64-bit BAR. This needs a little bit of explanation. PCI bus protocol actually supports 32-bit and 64-bit memory space. The 64-bit memory space is supported through "dual cycle" addressing, i.e., access to the 64-bit address requires two cycles instead of one because the PCI bus is natively 32-bit bus in its implementation. However, we're not going to delve deeper into it because we are only concerned with the 32-bit PCI bus in this article. If you ask: Why only 32-bit? It's because the

CPU we are talking about here only supports 32-bit operation—without physical address extension (PAE), and we don't deal with PAE here.

Figure 7 shows that bit 3 controls the *prefetching* in BAR that maps to CPU memory space. Prefetching in this context means that the CPU fetches the contents of memory addressed by the BAR before a request to that specific memory address is made, i.e., the "fetching" happens in advance, hence "pre"-fetching. This feature is used to improve the overall PCI device memory read speed.

Figure 6 and Figure 7 shows the rest of the BAR contents as "Base Address." Not all of the bits in the part marked as "Base Address" are writeable. The number of writeable bits depends on the size of the memory range required by the PCI device onboard memory. Detail of which bits of the "Base Address" are hardcoded and which bits are writeable is discussed in the next section: PCI BAR Sizing Implementation Note.

Figure 8 XROMBAR Format

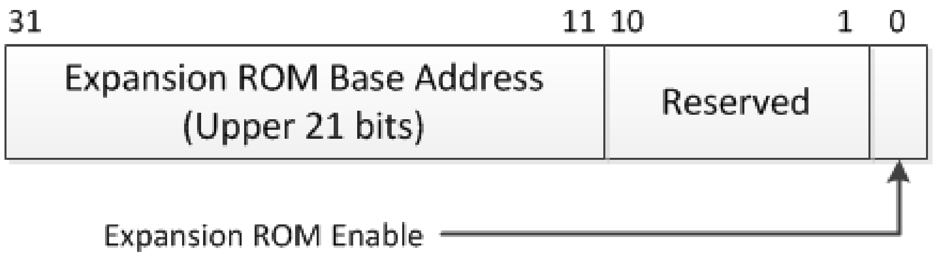


Figure 8 shows the XROMBAR format. As you can see, the lowest 11 bits are practically hardcoded to zero—because the expansion ROM enable bit doesn't act as an address bit, only as a control bit. This means a PCI expansion ROM must be mapped to a 2 KB boundary. The "expansion ROM enable" bit controls whether access to the PCI expansion ROM is enabled or not. When the bit is set to one, the access is enabled and when the bit is set to zero the access is disabled. If you look at Figure 3 "PCI Configuration Registers Type 0," you see the command register at offset `04h`. There's one bit in the command register called the *memory space* bit that must also be enabled to enable access to the PCI

pansion

nand

be set to

The number of writeable bits in the "expansion ROM base address" bits depends on the size of the memory range required by the PCI expansion ROM. Detail of which bits of the “expansion ROM base address” are hardcoded and which bits are writeable is discussed in the next section: PCI BAR Sizing Implementation Note. The process for BAR initialization also applies to initializing the XROMBAR.

In practice, some vendors prefer to use BAR instead of XROMBAR to map the PCI expansion ROM to either the CPU memory space or the CPU I/O space. One particular example is the old Realtek RTL8139D LAN card chip. This chip maps the PCI expansion ROM to the CPU I/O space instead of the CPU memory space by using a BAR, instead of the XROMBAR. You can see the details in the chip datasheet. This shows that, despite the presence of the bus protocol standard, some vendors prefer a quite different approach compared to what the standard suggests. Well, the Realtek approach is certainly not wrong as per the PCI bus protocol but it’s quite outside of the norm. However, I think there might be an economic reason to do so. Perhaps, by doing that Realtek saved quite sizeable die area in the LAN card chip. This translated into cheaper production costs.

PCI BAR Sizing Implementation Note

At this point, the PCI BAR format and XROMBAR format are already clear except for the base address bits, which as stated previously, contain device specific values. Base address bits depend on the size of the memory range required by the PCI device. The size of the memory range required by a PCI device is calculated from the number of writeable bits in the base address bits part of the BAR. The detail is described in the BAR sizing implementation note in PCI specification v2.3, as follows:

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

Decoding (I/O or memory) of a register is disabled via the command register (offset 04h in PCI configuration space) before sizing a base address register. Software saves the original value of the base address register, writes 0FFFFFFFFh to the register, then (the software) reads it back. Size calculation can be done from the 32-bit value read by first clearing the 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 return zero upon read. The original value in the base address register is restored before re-enabling decode in the command register (offset 04h in PCI configuration space) 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.

The implementation note above is probably still a bit vague. Now, let's see how the BAR in the AGP video card chip should behave in order to allocate 32 mb from the CPU memory space for the onboard video memory. The address range allocation happens during "BAR sizing" phase in the BIOS execution. "BAR sizing" routine is part of the BIOS code that builds the system address map. The "BAR sizing" happens in these steps (as per the implementation note above):

1. BIOS code saves the original value of the video card chip BAR to temporary variable in RAM. There are six BARs in the PCI configuration registers of the video card chip. We assume the BIOS work with the first BAR (offset 10h in the PCI configuration registers).
2. BIOS code writes `FFFF_FFFFh` to the first video card chip BAR. The video card chip would allow write only to writeable bits in the Base Address part of the BAR. The lowest 4-bits are hardcoded values meant as control bits for the BAR. The bits controls prefetching behavior, whether the BAR is 32-bit or 64-bit and whether the BAR maps to CPU memory space or CPU I/O space. Here we assume that the BAR maps to CPU memory space.
3. The BAR should contain `FE00_000Xh` after the write in step 2—because the video card size is 32 mb, you will see later how this value transforms into 32 mb. `X` denotes variable contents that we are not interested in, because it depends on the specific video card and it doesn't affect "BAR sizing" process.
4. As per the implementation note, the lowest 4 bits must be reset to `0h`. Therefore, now we have `FE00_0000h`.
5. Then, the value obtained in the previous step must be inverted (logical NOT 32 bits). Now, we have `01FF_FFFFh`.
6. Then, the value obtained in the previous step must be incremented by one. Now, we have `200_0000h`, which is 32 mb, the size of memory range required by the AGP video card.

Perhaps, you're asking why the `FFFF_FFFFh` writes to the BAR only returns `FE00_000Xh`? The answer is because only the top seven bits of the BAR are writeable. The rest is hardcoded to zero, except the lowest 4 bits, which are used for BAR information—I/O or memory mapping indicator, prefetching, and 32-bit/64-bit indicator. These lowest 4 bits could be hardcoded to non-zero values, which is why the read result must be reset to zero. It's possible to relocate the video card memory anywhere in the 4GB CPU memory space in a 32 mb boundary because the top seven bits of the 32-bit BAR are writeable—I believe you can do the math yourself for this.

After the "BAR sizing," the BIOS knows the amount of memory required by the video card, i.e., 32 mb. The BIOS can now map the video card memory into the CPU memory space by writing the intended address into the video card BAR. For example, to map the video card memory to address 256 mb, the BIOS would write `1000_0000h` (256 mb) to the video card BAR; with this value, the top seven bits of the BAR contains the `0001_000b` binary value.

Now, you might be asking, what about XROMBAR? Well, the process just the same, except that XROMBAR maps read only memory (ROM) instead of RAM—as in the video card memory sample explained here—into the CPU memory space. This means there's no prefetching of bits to be read or processed during the "BAR sizing" because ROM usually doesn't support prefetching at all unlike RAM, which acts as video card memory. The explanation on "BAR sizing" in this subsection should clarify the expansion ROM mapping to the system address map, which is not very clear in my previous article (http://resources.infosecinstitute.com/pci-expansion-rom/).

The AGP Graphics Address Remapping/Relocation Table (GART)

The example of PCI device mapping to system address map in this article is an AGP video card. Logically, an AGP is actually an "enhanced" PCI device and it has some peculiarities, such as the fact that the AGP device slot runs at 66 MHz instead of 33 MHz in ordinary PCI slot and AGP also has some AGP-specific logic support inside the northbridge chipset.

One of the AGP-specific supports in the northbridge is the so-called AGP graphics address remapping/relocation table (GART) hardware logic. AGP GART —or GART for short—allows the video card chip in the video card to use a portion of the main memory (RAM) as video memory if the application(s) that use the video memory runs out of the onboard video memory. The GART hardware is basically a sort of memory management unit (MMU) that presents chunks of the RAM—allocated as additional video memory—as one contiguous memory to the video card chip. This is required because, by the time the video card chip requests for additional video memory to the system, the RAM is very probably already fragmented because other applications have used portions of the RAM for their own purposes.

The GART hardware has a direct relation to the system address map initialization via the so-called AGP aperture. For those old enough, you might remember setting the AGP aperture size in your PC BIOS back then. The GART in some respect is a precursor to the input/output memory management unit (IOMMU) technology that is in use in some of present-day hardware.

Now, let's look deeper into the GART and the AGP aperture. Figure 9 shows how the GART and the AGP aperture work.

Figure 9 AGP GART and AGP Aperture in the System Address Map

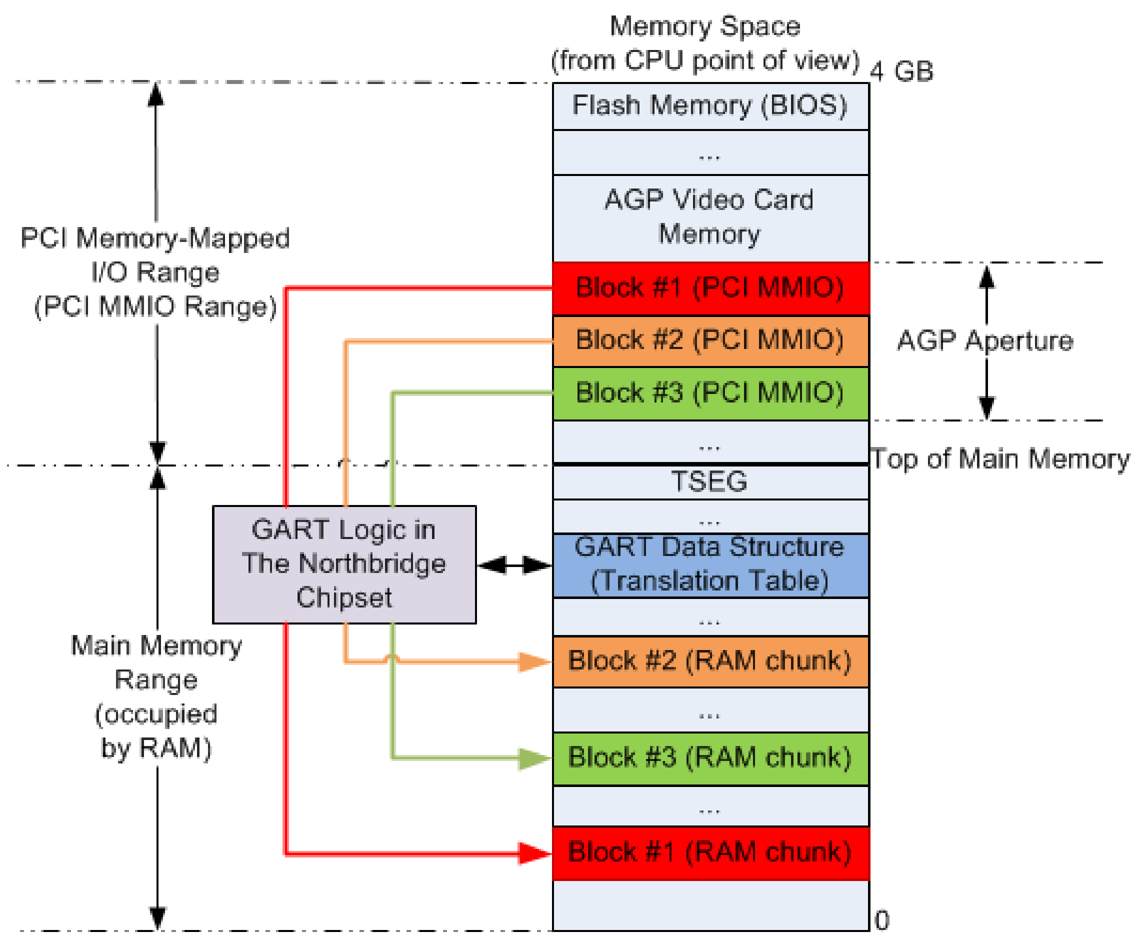


Figure 9 shows implementation of the AGP aperture and GART. The AGP aperture—which is set in the BIOS/platform firmware—basically reserves a contiguous portion of the PCI MMIO range to be used as additional video memory in case the video memory is exhausted at runtime (inside the OS). The size of the AGP aperture is set in the BIOS setting. Therefore, when the BIOS initializes the system address map, it sets aside a contiguous portion of the PCI MMIO range to be used as the AGP aperture.

The GART logic in the northbridge chipset contains a register that contains pointer to the GART data structure (translation table) in RAM. The GART data structure is stored in RAM, much like the global descriptor table or local descriptor table in x86/x64 CPUs. The video driver working alongside the OS kernel initializes the GART data structure. When the video memory is running low, the video driver (working alongside the OS kernel) allocates chunks of RAM as required.

The video card chip accesses the additional video memory via the AGP aperture. Therefore, the video card chip only sees a contiguous additional video memory instead of chunks of memory in RAM. The GART hardware in the northbridge chipset translates from the AGP aperture “address space” to the RAM chunk’s “address space” for the video card chip. For example, in Figure 9, when the video card chip accesses block #1 in the AGP aperture region, the GART logic would translate the access into access to block #1 in the corresponding RAM chunk—the block is marked in red in Figure 9. The GART logic translates the access based on the contents of the GART data structure, which is also located in RAM but cached in the GART logic, much like the descriptor cache in x86/x64 CPUs.

The downside of the AGP architecture lies in the GART being located in the northbridge, a shared and critical resource for the entire system. This implies that a misbehaving video card driver could crash the entire system because the video card literally programmed the northbridge for GART-related stuff. This fact encourages the architects of the PCIe protocol to require the GART logic to be implemented in the video card chip instead of in the chipset in systems that implement the PCIe protocol. For details on the requirement, you can read this: http://msdn.microsoft.com/en-us/library/windows/hardware/gg463285.aspx.

Now you should have a very good overall view of the effect of the AGP in the system address map. The most important thing to remember is that the GART logic consults a translation table, i.e., the GART data structure, in order to access the real contents of the additional video memory in RAM. A similar technique is employed in IOMMU—the use of translation table.

Hijacking BIOS Interrupt 15h AX=E820h Interface

Now let’s see how you can query the system for the system address map. In legacy systems with legacy platform firmware, i.e., BIOS, the most common way to request system address map is via interrupt 15h function E820h (ax=E820h).

The interrupt must be performed when the x86/x64 system runs in real mode; right after the BIOS completes platform initialization. You can find details of this interface at: http://www.uruk.org/orig-grub/mem64mb.html. Interrupt 15h function E820h is sometimes called the E820h interface. We'll adopt this naming here.

A legacy boot rootkit—let's call it bootkit—could hide in the system by patching the interrupt 15h, function E820h handler. One of the ways to do that is to alter the *address range descriptor* structure returned by the E820h interface. The *address range descriptor* structure is defined as follows:

```
typedef AddressRangeDescriptorTag{
    unsigned long BaseAddrLow;
    unsigned long BaseAddrHigh;
    unsigned long LengthLow;
    unsigned long LengthHigh;
    unsigned long Type;
} AddressRangeDescriptor;
```

The `type` field in the *address range descriptor* structure determines whether the memory range is available to be used by the OS, i.e. can be read and written by the OS. The encoding of the type field as follows:

- A value of 1 means this run (the returned memory range from the interrupt handler) is available RAM usable by the operating system.
- A value of 2 means this run of addresses (the returned memory range from the interrupt handler) is in use or reserved by the system, and must not be used by the operating system.
- "Other values" means undefined—reserved for future use. Any range of this type must be treated by the OS as if the type returned was reserved (the return value is 2).

If the RAM chunk used by the bootkit is marked as reserved region in the `Type` field of the *address range descriptor* structure, it means that the OS would regard the RAM chunk as "off-limits." This practically "hides" the bootkit from the OS. This is one of the ways a bootkit can hide in a legacy system.

Anyway, memory range—in the CPU memory space—consumed by PCI MMIO devices would also be categorized as a reserved region by the BIOS interrupt 15h function E820h handler because it's not in the RAM region and it should not be regarded as such. Moreover, random read and write to PCI devices can have unintended effects.

The users of the E820h interface are mostly the OS and the OS bootloader. The OS needs to know the system address map in order to initialize the system properly. The OS bootloader sometimes has additional function such as RAM tests and in some circumstances it also passes the system memory map to the OS. That's why the OS bootloader is also a user of the E820h interface.

UEFI: The GetMemoryMap() Interface

Now you might be asking: what is the equivalent of the E820h interface in UEFI? Well, the equivalent of the E820h interface in UEFI is the `GetMemoryMap()` function. This function is available as part of the UEFI boot services. Therefore, you need to traverse into the UEFI boot services table to "call" the function. The "simplified" algorithm to call this function as follows:

1. Locate the EFI system table.
2. Traverse to the `EFI_BOOTSERVICES_TABLE` in the EFI System Table.
3. Traverse the `EFI_BOOTSERVICES_TABLE` to locate the `GetMemoryMap()` function.
4. Call the `GetMemoryMap()` function.

The `GetMemoryMap()` function returns a data structure that is similar to the one returned by the legacy E820h interface. The data structure is called `EFI_MEMORY_DESCRIPTOR`. Well, this article doesn't try to delve deeper into this interface. You can read details of the interface and the `EFI_MEMORY_DESCRIPTOR` in the UEFI specification. Should you be interested in digging deeper, the `GetMemoryMap()` function is located in the boot services chapter of the UEFI specification, under the memory allocation services section.

Closing Thoughts

Thanks go to reader Jimbo Bob, who asked the question regarding clarification on PCI expansion ROM chip address mapping in my "Malicious PCI Expansion ROM" article (http://resources.infosecinstitute.com/pci-expansion-rom/). I was not aware of the insufficiency of my explanation in that article. Perhaps, because I have been working on BIOS and other stuff bare metal for years, I took it for granted that readers of my article would be sufficiently informed regarding the bus protocols I talked about. So here I am, fixing that mistake.

Looking forward, I'm preparing another article in the same spirit as this one that focuses on present-day bus protocol, the PCI express (PCIe). For those who wandered to bare metal programming in x86/x64 for the first time, I hope this article will help understanding the system. Hopefully the reader becomes fully informed about where to go to find information regarding chip-specific information, particularly, knowledgeable enough to travel the web to find relevant x86/x64-related chip/chipset datasheet according to his/her needs.

AUTHOR
Darmawan Salihun

Darmawan Salihun has been focusing on BIOS-related security research since 2002. He has published papers on BIOS reverse engineering, BIOS code injection techniques, BIOS hacking, Plug and Play BIOS, and PCI protocol exploitation in the CodeBreakers Journal. In 2006, he wrote the BIOS Disassembly Ninjutsu book which covers the result of his research on BIOS security. His research on BIOS and UEFI security continues to this day. He is also a researcher with InfoSec Institute.

Security+ Practice Exam

CEH Practice Exam

CISSP Practice Exam

EDITORS CHOICE

- Securely Managing Cloud Credentials
- APT28: Cybercrime or State-sponsored Hacking?
- Interview: Kalyan Ramanathan, VP of Product Marketing at AppDynamics

RELATED BOOT CAMPS

Information Security

CCNA

PMP

Microsoft

Incident Response

Information Assurance

8570

MORE POSTS BY AUTHOR

NSA Backdoor Part 2, BULLDOZER: And, Learn How to DIY a NSA Hardware Implant

NSA BIOS Backdoor a.k.a.

God Mode Malware Part 1: DEITYBOUNCE

System Address Map Initialization in x86/x64 Architecture Part 2: PCI Express-Based Systems

Securely Managing Cloud Credentials

APT28: Cybercrime or State-sponsored Hacking?

Interview: Kalyan Ramanathan, VP of Product...

Case Study: Evading Automated Sandbox –...

13 Comments **InfoSec Institute Resources** Login

Recommend Share Sort by Best

Join the discussion…

hispresencematters • 4 months ago

Thank you for writing this article. it is well written and comprehensive. I learned z lot from reading it.

• Reply • Share ›

Aytaç • 6 months ago

This is one of the best article i've ever read.

• Reply • Share ›

Gaurav • 9 months ago

Thanks for writing this article. It cleared lot of my doubts and helped me in understanding PCI better.

• Reply • Share ›

Sanjay • a year ago

A really helpful article to know PCI mapping which was very confusing for be before. Thanks a lot
how to access PCI config space by using just by using 2 port registers if we have multiple PCI devices available ? and how it will know how many PCI devices are present in the system?

• Reply • Share ›

Vikram Sharma • a year ago

Hello Darmawan Salihun.
I am looking for a way to fetch the Hardware specific Information (Vendor ID and Product ID) of Flash BIOS chip from the OS (Linux) level. Need your suggestions over this.

Thanks

• Reply • Share ›

Darmawan Salihun • a year ago

@Dicky and krishz: when the RAM size is 4GB and PAE is not in use, part of the RAM will be hidden and cannot be used/accessed because access to the range will be mapped to PCI devices. The PCI devices will occupy the CPU memory range above 3GB in most x86 PCI systems. This effectively creates unusable space in RAM (sometimes referred to as "range shadowed by PCI device") which cannot be accessed by any code because the "address router" in the chipset will forward access to that range to PCI device memory instead of RAM. If you read the second part of this article (http://resources.infosecinstit..., you'll see that those "shadowed range" is remapped to address above 4GB.

• Reply • Share ›

Dicky • a year ago

Nice thorough PCI article gan.

But maybe you could explain a little about bus number, device number, and function number before mentioning 0xCFC and 0xCFF ports. My point is, there must be an emphasize that by using just those two 32-bit ports, we can actually access every configuration register owned by each PCI device in the system.

Btw, I think you haven't yet answered krishz's second question:
"What will be the system memory map if 4GB RAM is used?"
Because I want to know as well :)
Assuming PAE is not used.

• Reply • Share ›

rajkumar • a year ago

Excellent information...i have learned lot...cleared many unanswered questions about PCI

• Reply • Share ›

Darmawan Salihun • a year ago

Hi,
1. PCI config space resides in the CPU I/O space. Port CF8h-CFBh (address) and Port CFCh-CFFh (data). These ports are the entry points. As for where they are implemented, each logical PCI device has its own configuration space registers. It's the job of the hostbridge (and possibly southbridge, depending on where the target PCI device is located) to route accesses to ports mentioned above to the target PCI device. Therefore, always keep in mind that overall PCI config space is actually "a pool" of the individual PCI devices configuration space registers. Unimplemented PCI register returns FFh on reads (as per PCI specs).

2. PCI config space itself doesn't use CPU memory space, but PCI device MMIO does use CPU memory space. You might be referring to PCI Express (PCIe). In that case, read the second part article: http://resources.infosecinstit...

• Reply • Share ›

krishz • a year ago

Hey,, I have read this article and also the article regarding malicious code in PCI exp ROM. Nice work !!!

I have few questions
1) Where does PCI configuration space really get stored? If it is CPU memory space,then what is the memory? Does it has enough memory space for having all devices PCI config space?...

2) What will be the system memory map if 4GB RAM is used?? Which will be TOM and How PCI configuration space is mapped in this case??

please clear me!!!

Thanks in Advance !!!

• Reply • Share ›

Dirk Van Aken • 2 years ago

I have been searching a lot to find a crisp explantion of the initialization principles behind PCI-Express BAR registers. Finally I found your excellent arcticle on this subject

!

Thank you for sharing your knowledge on this difficult matter !

• Reply • Share ›

Darmawan Salihun • 2 years ago

If you mean that "you learned more", then I'm glad about it ;-)

• Reply • Share ›

Maidulpci • 2 years ago

Learn more about PCI-Based Systems.

• Reply • Share ›

ALSO ON **INFOSEC INSTITUTE RESOURCES** WHAT'S THIS?

Stack Based Buffer Overflow in Win 32 Platform: The Basics

2 comments • 2 months ago

Avat Jennifer Rose — Simple and straight. Very helpful to clear basic facts. Thanks. :)

Windows Functions in Malware Analysis – Cheat Sheet – Part 1

1 comment • 9 days ago

Avat V — Such 1337. Much wow.

Child Porn, SSNs, and Gamergate: The Vile Tale of 8chan

5 comments • 3 months ago

Avat Julie — Depression actually has some noticeable differences between men and

TrueCrypt Security: Securing Yourself against Practical TrueCrypt Attacks

2 comments • 4 months ago

Avat rodrigosruiz — Paper about clone headers. https://www.researchgate.net/p...