Code Breakers Journal

Vol. 2, No. 1 (2005)
http://www.CodeBreakers-Journal.com

Low Cost Embedded x86 Teaching Tool

MS Darmawan*

* Corresponding Author

Received: 19. Apr. 2005, Accepted: 23. Apr. 2005, Published: 24. Feb. 2005

Abstract

The wide availability of personal computer based on the x86 architecture that conform to the PCI specification version 2.1 and Plug and Play BIOS specification version 1.0A or higher, along with the existence of free open source software development tools for this architecture, provides an opportunity to create a low cost embedded system teaching tool based on it. In this paper we will explain one of the implementation of this idea by exploiting the so called "Bootstrap Entry Vector" that exist as part of the Plug and Play BIOS.

Keywords: *Embedded, X86, Teaching Tool*

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

1. Introduction

The main obstacle of teaching embedded system development in various universities is the cost of the hardware used for development and possibly the cost of the software tool. The existence of free open source software development tools for many processor architecture in recent years has solved the problem of the cost of the software tool needed. However, the cost of the hardware remains quite high for college students to afford. Especially in developing nation like in Indonesia, where cost is the main issue. From cost point of view, the PC itself is still affordable for the students, since it's used for wide variety of task, not just embedded system development. Meanwhile, buying hardware for embedded system development board can easily cost more than an entry-level PC or refurbished PC.

In this paper we explore the possibility to develop a low cost tool for teaching embedded system in the x86 processor architecture based on the PCI expansion ROM. It will become a playing-ground for the students to learn embedded system development in x86 platform. The term PCI expansion ROM in this paper is the PCI firmware which is embedded in a PCI expansion card. It's also sometimes called PCI option ROM. We will use the term interchangeably. PCI expansion ROM has a broader meaning than the definition that has been mentioned above, but we are focusing on that type of expansion ROM in this paper. The reader might be aware that PCI expansion ROM can also be embedded inside the main board BIOS as a component. We are not considering the latter type of PCI expansion ROM here.

We are going to demonstrate the development of a custom PCI expansion ROM that is going to be embedded in "special" PCI expansion card by using free open source software development tool. This PCI expansion ROM and it's corresponding "special" PCI expansion card is the hardware-software complex that acts as the embedded system development teaching tool. The cost of this hardware-software complex can be very low (the cost of the PC is not included), from around Rp. 25.000,00 (around US $ 2.5) to nothing at all if we can find the PCI expansion card from junk yard.

2. Prerequisite

2.1. x86 Memory Map

Understanding of the x86 memory map is a must to be able to develop an embedded system based on this platform. We will start with the explanation of the booting process. An x86 CPU begins its execution at address FFFFFFF0h physical address[1]. This address is the address of the first instruction within the main board (system) bios. It's the responsibility of the main board chipset to remap this address into the system bios chip. The system bios is the very first program that the processor executes. Below is an explanation of the typical memory map of an x86 based system just after the system bios finished initialization.

In the memory map above, of particular interest is the expansion ROM area. We will be dealing with this area later as we are developing the custom PCI expansion ROM.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

<table>
<tr><th>Address Range</th><th>Detailed Explanation</th></tr>
<tr><td rowspan="5">Compatibility area
(0_0000h - F_FFFFh)</td><td>DOS Area (00000h–9FFFFh)
The DOS area is 640 KB in size and is always mapped to the main memory (RAM) by mainboard chipset</td></tr>
<tr><td>Legacy VGA Ranges and/or Compatible SMRAM Address Range (A0000h–BFFFFh)
The legacy 128-KB VGA memory range A0000h–BFFFFh (Frame Buffer) can be mapped to AGP or PCI Device. However, when compatible SMM space is enabled, SMM-mode processor accesses to this range are routed to physical system memory at this address. Non-SMM-mode processor accesses to this range are considered to be to the video buffer area as described above.</td></tr>
<tr><td>Expansion ROM Area (C0000h–DFFFFh)
This is the 128-KB ISA/PCI Expansion ROM region. The system BIOS copies PCI expansion ROM to this area (in RAM) from the corresponding PCI expansion card ROM chip and execute it from there. As for ISA expansion ROM, it only exist on system that support ISA expansion card and sometimes the expansion ROM chip of the corresponding card is "hardwired" certain memory range in this area. In most case, part of this memory range can be assigned one of four read/write states: read only, write only, read/write, or disabled. This state assigment is controlled by the setting of certain mainboard chipset registers. The system BIOS is responsible for assigning the correct read/write state.</td></tr>
<tr><td>Extended System BIOS Area (E0000h–EFFFFh)
This 64-KB area is divided into four, 16-KB segments. Each segment can be assigned independent read and write attributes so it can be mapped either to main memory or to the BIOS ROM chip via the system chipset. Typically, this area is used for RAM or ROM. On systems that only supports 64KB BIOS ROM chip capacity, this memory area always mapped to RAM.</td></tr>
<tr><td>System BIOS Area (F0000h–FFFFFh)
This area is a single, 64-KB segment. This segment can be assigned read and write attributes. It is by default (after reset) read/write disabled and cycles are forwarded to BIOS ROM chip via the system chipset. By manipulating the read/write attributes, the system chipset can “shadow” BIOS into the main memory. When disabled, this range is not remapped to main memory by the chipset.</td></tr>
<tr><td rowspan="2">Extended Memory Area
(10_0000h - FFFF_FFFFh)</td><td>Main system memory from 1 MB (10_0000h) to the Top of of RAM
This area can have a hole i.e. an area that is not mapped to RAM but mapped to ISA devices. This hole is dependent on the mainboard chipset configuration</td></tr>
<tr><td>AGP or PCI memory space from the Top of RAM to 4 GB (FFFF_FFFFh)
This area has 2 specific ranges :
APIC Configuration Space from FEC0_0000h (4 GB-20 MB) to FECF_FFFFh and FEE0_0000h to FEEF_FFFFh. This is also dependent on the mainboard chipset. Some chipset doesn't support APIC, hence this mapping doesn't exist.
High BIOS area from 4 GB to 4 GB – 2 MB. This address range mapped into the BIOS ROM chip. But it's dependent on the mainboard chipset. Some chipset only support mapping 4 GB - (4GB-256KB) for BIOS ROM chip. However, the 4 GB - (4GB-64KB) memory area is the least common denominator for all chipsets, this area is guaranteed to map into the BIOS ROM chip.
In most case, anything outside of these specific ranges but within the PCI memory space (top of RAM - 4GB) is mapped to PCI/AGP device that needs to map their "local memory" (memory local to the PCI card) to system memory space. This mapping is normally initialized by system BIOS. Access to this memory space is routed by the system chipset (memory controller). In the case of AMD Athlon64 and Opteron platform, this routing is handled by the processor itself, since the memory controller is embedded in the processor itself [2].</td></tr>
</table>

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

2.2. PnP BIOS Architecture

In this section, we are not going to provide a complete explanation of the PnP BIOS architecture. We will only explain parts of the PnP BIOS architecture that are needed to develop our hardware-software complex. A more thorough explanation regarding the system BIOS can be found in Award BIOS Reverse Engineering Paper [3].

The parts of PnP BIOS that are important to our project are the initialization of expansion/option ROM, i.e. initialization code that resides in the expansion cards and the bootstrap process, i.e. transferring control from BIOS to operating system after the BIOS has done initializing the system. Initialization of option ROM is part of the POST (Power-On Self Test) routine in the system BIOS. The related information from the PnP BIOS Specification 1.0A [5] are provided below.

2.2.1. POST Execution flow

The following steps outline a typical flow of a Plug and Play system BIOS POST. All of the standard ISA functionality has been eliminated for clarity in understanding the Plug and Play POST enhancements.

Step 1 **Disable all configurable devices**
Any configurable devices known to the system BIOS should be disabled early in the POST process.

Step 2 **Identify all Plug and Play ISA devices**
Assign Card Select Numbers (CSNs) to Plug and Play ISA devices but keep devices disabled. Also determine which devices are boot devices.

Step 3 **Construct an initial resource map of allocated resources**
Construct a resource map of resources that are statically allocated to devices in the system. If the system software has explicitly specified the system resources assigned to ISA devices in the system through the Set Statically Allocated Resource Information function, the system BIOS will create an initial resource map based on this information. If the BIOS implementation provides support for saving the last working configuration and the system software has explicitly assigned system resources to specific devices in the system, then this information will be used to construct the resource map. This information will also be used to configure the devices in the system.

Step 4 **Enable Input and Output Devices**
Select and enable the Input and Output Device. Compatibility devices in the system that are not configurable always have precedence. For example, a standard VGA adapter would become the primary output device. If configurable Input and Output Devices exists, then enable these devices at this time. If Plug and Play Input and Output Devices are being selected, then initialize the option ROM, if it exists, using the Plug and Play option ROM initialization procedure.

Step 5 **Perform ISA ROM scan**
The ISA ROM scan should be performed from C0000h to EFFFFh on every 2K boundary. Plug and Play Option ROMs are disabled at this time (except input and output boot devices) and will not be included in the ROM scan.

Step 6 **Configure the Initial Program Load(IPL) device**
If a Plug and Play device has been selected as the IPL device, then use the Plug and Play Option ROM procedure to initialize the device. If the IPL device is known to the system BIOS, then ensure that interrupt 19h is still controlled by the system BIOS. If not, recapture interrupt 19h and save the vector.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Step 7 **Enable Plug and Play ISA and other Configurable Devices**
If a static resource allocation method is used, then enable the Plug and Play ISA cards with conflict free resource assignments. Initialize the option ROMs and pass along the defined parameters. All other configurable devices should be enabled, if possible, at this time. If a dynamic resource allocation method is used, then enable the bootable Plug and Play ISA cards with conflict free resource assignments and initialize the option ROMs.

Step 8 **Initiate the Interrupt 19H IPL sequence**
Start the bootstrap loader. If the operating system fails to load and a previous ISA option ROM had control of the interrupt 19h vector, then restore the interrupt 19h vector to the ISA option ROM and re-execute the Interrupt 19h bootstrap loader.

Step 9 **Operating system takes over resource management**
If the loaded operating system is Plug and Play compliant, then it will take over management of the system resources. It will use the runtime services of the system BIOS to determine the current allocation of these resources. It is assumed that any unconfigured Plug and Play devices will be configured by the appropriate system software or the Plug and Play operating system.

2.2.2. Option ROM Support

This section outlines the Plug and Play Option ROM requirements. This Option ROM support is directed specifically towards boot devices; however, the Static Resource Information Vector permits non-Plug and Play devices which have option ROMs to take advantage of the Plug and Play Option ROM expansion header to assist a Plug and Play environment whether or not it is a boot device. *A boot device is defined as any device which must be initialized prior to loading the Operating System.* Strictly speaking, the only required boot device is the **Initial Program Load (IPL)** device upon which the operating system is stored. However, the definition of boot devices is extended to include a primary Input Device and a primary Output device. In some situations these I/O devices may be required for communication with the user. All new Plug and Play devices that support Option ROMs should support the Plug and Play Option ROM Header. In addition, all non-Plug and Play devices may be "upgraded" to support the Plug and Play Option ROM header as well. While these static ISA devices will still not have software configurable resources, the Plug and Play Option ROM Header will greatly assist a Plug and Play System BIOS in identification and selection of the primary boot devices. It is important to note that the Option ROM support outlined here is defined specifically for computing platforms based on the Intel X86 family of microprocessors and may not apply to systems based on other types of microprocessors.

2.2.2.1. Option ROM Header

The Plug and Play Option ROM Header follows the format of the Generic Option ROM Header extensions . The Generic Option ROM header is a mechanism whereby the standard ISA Option ROM header may be expanded with minimal impact upon existing Option ROMs. The pointer at offset 1Ah may point to ANY type of header. Each header provides a link to the next header; thus, future Option ROM headers may use this same generic pointer and still coexist with the Plug and Play Option ROM header. Each Option ROM header is identified by a unique string. The length and checksum bytes allow the System BIOS and/or System Software to verify that the header is valid. Standard Option ROM Header:

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Offset	Length	Value	Description	Type
0h	2h	AA55h	Signature	Standard
2h	1h	Varies	Option ROM Length	Standard
3h	4h	Varies	Initialization Vector	Standard
7h	13h	Varies	Reserved	Standard
1Ah	2h	Varies	Offset to Expansion Header Structure	New for Plug and Play

1. **Signature** - All ISA expansion ROMs are currently required to identify themselves with a signature WORD of AA55h at offset 0. This signature is used by the System BIOS as well as other software to identify that an Option ROM is present at a given address.
2. **Length** - The length of the option ROM in 512 byte increments.
3. **Initialization vector** - The system BIOS will execute a FAR CALL to this location to initialize the Option ROM. A Plug and Play System BIOS will identify itself to a Plug and Play Option ROM by passing a pointer to a Plug and Play Identification structure when it calls the Option ROM's initialization vector. If the Option ROM determines that the System BIOS is a Plug and Play BIOS, the Option ROM should not hook the input, display, or IPL device vectors (INT 9h, 10h, or 13h) at this time. Instead, the device should wait until the System BIOS calls the Boot Connection vector before it hooks any of these vectors.
   Note: A Plug and Play device should never hook INT 19h or INT 18h until its Boot Connection Vector, offset 16h of the Expansion Header Structure (section 3.2), has been called by the Plug and Play system BIOS.
   If the Option ROM determines that it is executing under a Plug and Play system BIOS, it should return some device status parameters upon return from the initialization call. See the section on Option ROM Initialization for further details.
   The field is four bytes wide even though most implementations may adhere to the custom of defining a simple three byte NEAR JMP. The definition of the fourth byte may be OEM specific.
4. **Reserved** - This area is used by various vendors and contains OEM specific data and copyright strings.
5. **Offset to Expansion Header** - This location contains a pointer to a linked list of Option ROM expansion headers. Various Expansion Headers (regardless of their type) may be chained together and accessible via this pointer. The offset specified in this field is the offset from the start of the option ROM header.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

2.2.2.2. Expansion Header for Plug and Play

Offset	Length	Value	Description	
0h	4 BYTES	$PnP (ASCII)	Signature	Generic
04h	BYTE	Varies	Structure Revision	01h
05h	BYTE	Varies	Length (in 16 byte increments)	Generic
06h	WORD	Varies	Offset of next Header (0000h if none)	Generic
08h	BYTE	00h	Reserved	Generic
09h	BYTE	Varies	Checksum	Generic
0Ah	DWORD	Varies	Device Identifier	PnP Specific
0Eh	WORD	Varies	Pointer to Manufacturer String (Optional)	PnP Specific
10h	WORD	Varies	Pointer to Product Name String (Optional)	PnP Specific
12h	3 BYTE	Varies	Device Type Code	PnP Specific
15h	BYTE	Varies	Device Indicators	PnP Specific
16h	WORD	Varies	Boot Connection Vector - Real/Protected mode (0000h if none)	PnP Specific
18h	WORD	Varies	Disconnect Vector - Real/Protected mode (0000h if none)	PnP Specific
1Ah	**WORD**	**Varies**	***Bootstrap Entry Point - Real/Protected mode (0000h if none)***	**PnP Specific**
1Ch	WORD	0000h	Reserved	PnP Specific
1Eh	WORD	Varies	Static Resource Information Vector- Real/Protected mode (0000h if none)	PnP Specific

Signature - All Expansion Headers will contain a unique expansion header identifier. The Plug and Play expansion header's identifier is the ASCII string "$PnP" or hex 24 50 6E 50h (Byte 0 = 24h ... Byte 3 = 50h).

Structure Revision - This is an ordinal value that indicates the revision number of this structure only and does not imply a level of compliance with the Plug and Play BIOS version.

Length - Length of the entire Expansion Header expressed in sixteen byte blocks. The length count starts at the Signature field.

Offset of Next Header - This location contains a link to the next expansion ROM header in this Option ROM. If there are no other expansion ROM headers, then this field will have a value of 0h. The offset specified in this field is the offset from the start of the option ROM header.

Reserved - Reserved for Expansion

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Checksum - Each Expansion Header is checksummed individually. This allows the software which wishes to make use of an expansion header (in this case, the system BIOS) the ability to determine if the expansion header is valid. The method for validating the checksum is to add up all byte values in the Expansion Header, including the Checksum field, into an 8-bit value. A resulting sum of zero indicates a valid checksum operation.

Device Identifier - This field contains the Plug and Play Device ID.

Pointer to Manufacturer String (Optional) - This location contains an offset relative to the base of the Option ROM which points to an ASCIIZ representation of the board manufacturer's name. This field is optional and if the pointer is 0 (Null) then the Manufacturer String is not supported.

Pointer to Product Name String (Optional) - This location contains an offset relative to the base of the Option ROM which points to an ASCIIZ representation of the product name. This field is optional and if the pointer is 0 (Null) then the Product Name String is not supported.

Device Type Code - This field contains general device type information that will assist the System BIOS in prioritizing the boot devices. The Device Type code is broken down into three byte fields. The byte fields consist of a Base-Type code that indicates the general device type. The second byte is the device Sub-Type and its definition is Plug and Play BIOS Specification 1.0A Page 18 dependent upon the Base-Type code. The third byte defines the specific device programming interface, IF.-Type, based on the Base-Type and Sub-Type.
Refer to Plug and Play BIOS Specification 1.0A Appendix B for a description of Device Type Codes.

Device Indicators - This field contains indicator bits that identify the device as being capable of being one of the three identified boot devices: Input, Output, or Initial Program Load (IPL).

Bit	Description
7	A 1 indicates that this ROM supports the Device Driver Initialization Model
6	A 1 indicates that this ROM may be Shadowed in RAM
5	A 1 indicates that this ROM is Read Cacheable
4	A 1 indicates that this option ROM is only required if this device is selected as a boot device.
3	Reserved (0)
2	A 1 in this position indicates that this device is an Initial Program Load (IPL) device.
1	A 1 in this position indicates that this device is an Input device.
0	A 1 in this position indicates that this device is a Display device.

Boot Connection Vector (Real/Protected mode) - This location contains an offset from the start of the option ROM header to a routine that will cause the Option ROM to hook one or more of the primary input, primary display, or Initial Program Load (IPL) device vectors (INT 9h, INT 10h, or INT 13h), depending upon the parameters passed during the call.

When the system BIOS has determined that the device controlled by this Option ROM will be one of the boot devices (the Primary Input, Primary Display, or IPL device), the System ROM will execute a FAR CALL to the location pointed to by the Boot Connection Vector. The system ROM will pass the following parameters to the options ROM's Boot Connection Vector:

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Reg On Entry	Description
AX	Provides an indication as to which vectors should be hooked by specifying the type of boot device this device has been selected as. Bit 7..3 Reserved(0) Bit 2 1=Connect as IPL (INT 13h) Bit 1 1=Connect as primary Video (INT 10h) Bit 0 1=Connect as primary Input (INT 09h)
ES:DI	Pointer to System BIOS PnP Installation Check Structure (See section 4.4)
BX	CSN for this card, ISA PnP devices only. If not an ISA PnP device then this parameter will be set to FFFFh.
DX	Read Data Port, (ISA PnP devices only. If no ISA PnP devices then this parameter will be set to FFFFh.

Disconnect Vector (Real/Protected mode) - This vector is used to perform a cleanup from an unsuccessful boot attempt on an IPL device. The system ROM will execute a FAR CALL to this location on IPL failure.

Bootstrap Entry Vector (Real/Protected mode) *- This vector is used primarily for* ***RPL (Remote Program Load)*** *support. To RPL (bootstrap), the System ROM will execute a FAR CALL to this location. The System ROM will call the Real/Protected Mode Bootstrap Entry Vector instead of INT 19h if:*
a) The device indicates that it may function as an IPL device,
b) The device indicates that it does not support the INT 13h Block Mode interface,
c) The device has a non-null Bootstrap Entry Vector,
d) The Real/Protected Mode Boot Connection Vector is null.
The method for supporting RPL is beyond the scope of this specification. A separate specification should define the explicit requirements for supporting RPL devices.

Reserved - Reserved for Expansion

Static Resource Information Vector - This vector may be used by non-Plug and Play devices to report static resource configuration information. Plug and Play devices should not support the Static Resource Information Vector for reporting their configuration information. This vector should be callable both before and/or after the option ROM has been initialized. The call interface for the Static Resource Information Vector is as follows:

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Entry: ES:DI

Pointer to memory buffer to hold the device's static resource configuration information. The buffer should be a minimum of 1024 bytes. This information should follow the System Device Node data structure, except that the Device node number field should always be set to 0, and the information returned should only specify the currently allocated resources (Allocated resource configuration descriptor block) and not the block of possible resources (Possible resource configuration descriptor block). The Possible resource configuration descriptor block should only contain the END_TAG resource descriptor to indicate that there are no alternative resource configuration settings for this device because the resource configuration for this device is static. Refer to the Plug and Play ISA Specification under the section labeled Plug and Play Resources for more information about the resource descriptors. This data structure has the following format:

Field	Size
Size of the device node	WORD
Device node number/handle	BYTE
Device product identifier	DWORD
Device type code	3 BYTES
Device node attribute bit-field	WORD
Allocated resource configuration descriptor block	VARIABLE
Possible resource configuration descriptor block - should only specify the END_TAG resource descriptor	2 BYTES
Compatible device identifiers	VARIABLE

Refer to section 4.2 for a complete description of the elements that make up the System Device Node data structure.

For example, an existing, non-Plug and Play SCSI card vendor could choose to rev the SCSI board's Option ROM to support the Plug and Play Expansion Header. While this card wouldn't gain any of the configuration benefits provided to full hardware Plug and Play cards, it would allow Plug and Play software to determine the devices configuration and thus ensure that Plug and Play cards will map around the static SCSI board's allocated resources.

2.2.2.3. Option ROM Initialization

The System BIOS will determine if the Option ROM it is about to initialize supports the Plug and Play interface by verifying the Structure Revision number in the device's Plug and Play Header Structure. For all Option ROMs compliant with the 1.0 Plug and Play BIOS Specification, the System BIOS will call the device's initialization vector with the following parameters:

Reg On Entry	Description
ES:DI	Pointer to System BIOS PnP Installation Check Structure (See section 4.4)
BX	CSN for this card, ISA PnP devices only. If not an ISA PnP device then this parameter will be set to FFFFh
DX	Read Data Port, (ISA PnP devices only. If no ISA PnP devices then this parameter will be set to FFFFh.

For other bus architectures refer to the appropriate specification for register parameters on entry. During initialization, a Plug and Play Option ROM may hook any vectors and update any data structures required for it to access any attached devices and perform the necessary identifications and

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

initializations. However, upon exit from the initialization call, the Option ROM must restore the state of any vectors or data structures related to boot devices (INT 9h, INT 10h, INT 13h, and associated BIOS Data Area [BDA] and Extended BIOS Data Area [EBDA] variables).

Upon exit from the initialization call, Plug and Play Option ROMs should return some boot device status information in the following format:

Return Status from Initialization Call:

AX Bit	Description
8	1 = IPL Device supports INT 13h Block Device format
7	1 = Output Device supports INT 10h Character Output
6	1 = Input Device supports INT 9h Character Input
5:4	00 = No IPL device attached 01 = Unknown whether or not an IPL device is attached ***10 = IPL device attached (RPL devices have a connection).*** 11 = Reserved
3:2	00 = No Display device attached 01 = Unknown whether or not a Display device is attached 10 = Display device attached 11 = Reserved
1:0	00 = No Input device attached 01 = Unknown whether or not an Input device is attached 10 = Input device attached 11 = Reserved

2.2.2.4. Option ROM Initialization flow

The following outlines the typical steps used to initialize Option ROMs during a Plug and Play system BIOS POST:

Step 1 **Initialize the boot device option ROMs.**
This includes the Primary Input, Primary Output, and Initial Program Load (IPL) device option ROMs.

Step 2 **Initialize ISA option ROMs by performing ISA ROM scan**
The ISA ROM scan should be performed from C0000h to EFFFFh on every 2k boundary. Plug and Play option ROMs will not be included in the ROM scan.

Step 3 **Initialize option ROMs for ISA devices which have a Plug and Play option ROM.**
Typically, these devices will not provide support for dynamic configurability. However, the resources utilized by these devices can be obtained through the Static Resource Information Vector as described in section 3.2.

Step 4 **Initialize option ROMs for Plug and Play cards which have a Plug and Play option ROM.**

Step 5 **Initialize option ROMs which support the Device Driver Initialization Model (DDIM).**
Option ROMs which follow this model make the most efficient use of space consumed by option ROMs. Refer to Appendix B for more information on the DDIM.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Up to this point we have known that the facility of the PnP BIOS that will help us in developing our teaching tool is the option ROM and it's corresponding **Bootstrap Entry Vector (BEV)**. The reason for selecting this bootstrap mechanism is: *the core functionality of the PC that will be used must not be disturbed by the the new functionality of the PC as the embedded system development tool and target platform. In other word, by setting up the Option ROM to behave as RPL device, the Option ROM will only be executed as the bootstrap device if the RPL i.e. Boot From LAN support is activated in the system BIOS. By doing things this way, we can switch back and forth between normal usage of the PC and the usage of the PC as embedded system development target platform by setting the appropriate system BIOS setting, i.e. the Boot From LAN Activation entry.*

Later, we well demonstrate how to implement this logic by developing a custom option ROM that can be flashed into a real PCI LAN card or another type of PCI expansion card that is "hacked" to behave as is if it's a real LAN card from PnP BIOS point of view.

2.3. PCI PnP Expansion ROM Architecture

The PCI specification version 2.1 [4] explains that there are two types of PCI devices, i.e. PCI-to-PCI bridge device and Non PCI-to-PCI bridge device. This paper only deals with Non PCI-to-PCI bridge device. Eventhough this classification exist, there is one common property that all PCI device inherit, i.e. all PCI device has a predefined 256 byte hardware registers in its chip that is called PCI Configuration Space Header. This header differ quite significantly between PCI-to-PCI bridge device (type 01h header) and Non PCI-to-PCI bridge device (type 00h header). The header is used for many purposes, but the main usage is for configuring the corresponding PCI device. The "type 00h" header layout is provided below :

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 Register 0				10h
Base Address Register 1				14h
Base Address Register 2				18h
Base Address Register 3				1Ch
Base Address Register 4				20h
Base Address Register 5				24h
Cardbus CIS Pointer				28h
Subsystem ID		Subsystem Vendor ID		2Ch
Expansion ROM Base Address (XROMBAR)				30h
Reserved				34h
Reserved				38h
Max_Lat	Min_Gnt	Interrupt Pin	Interrupt Line	3Ch

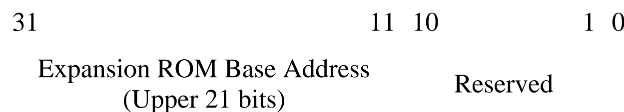
Type 00h Configuration Space Header

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

As we can see in this header layout, there exist a special register that handle expansion ROM addressing. Below is an explanation about the Expansion ROM Base Address Register (**XROMBAR**) and the related Expansion ROM from the PCI specification [4].

2.3.1. Expansion ROM Base Address Register

Some PCI devices, especially those that are intended for use on add-in modules in PC architectures, require local EPROMs for expansion ROM. The four-byte register at offset 30h in a type 00h predefined header is defined to handle the base address and size information for this expansion ROM. The figure below shows how this word is organized. The register functions exactly like a 32-bit Base Address register except that the encoding (and usage) of the bottom bits is different. The upper 21 bits correspond to the upper 21 bits of the Expansion ROM base address. The number of bits (out of these 21) that a device actually implements depends on how much address space the device requires. For instance, a device that requires a 64 KB area to map its expansion ROM would implement the top 16 bits in the register, leaving the bottom 5 (out of these 21) hardwired to 0. Devices that support an expansion ROM must implement this register. Device independent configuration software can determine how much address space the device requires by writing a value of all 1's to the address portion of the register and then reading the value back. The device will return 0's in all don't-care bits, effectively specifying the size and alignment requirements. The amount of address space a device requests must not be greater than 16 MB.



Expansion ROM Base Address Register Layout

Bit 0 in the register is used to control whether or not the device accepts accesses to its expansion ROM. When this bit is 0, the device's Expansion ROM address space is disabled. When the bit is 1, address decoding is enabled using the parameters in the other part of the base register. This allows a device to be used with or without an expansion ROM depending on system configuration. The Memory Space bit in the Command register has precedence over the Expansion ROM enable bit. A device must respond to accesses to its expansion ROM only if both the Memory Space bit and the Expansion ROM Base Address Enable bit are set to 1. This bit's state after RST# is 0. In order to minimize the number of address decoders needed on a device, it may share a decoder between the Expansion ROM Base Address register and other Base Address registers.[41] When expansion ROM decode is enabled, the decoder is used for accesses to the expansion ROM and device independent software must not access the device through any other Base Address registers.

[41]Note that it is the address decoder that is shared, not the registers themselves. The Expansion ROM Base Address register and other Base Address registers must be able to hold unique values at the same time.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

2.3.2. PCI Expansion ROMs

The PCI specification provides a mechanism where devices can provide expansion ROM code that can be executed for device-specific initialization and, possibly, a system boot function. The mechanism allows the ROM to contain several different images to accommodate different machine and processor architectures. This section specifies the required information and layout of code images in the expansion ROM. Note that PCI devices that support an expansion ROM must allow that ROM to be accessed with any combination of byte enables. This specifically means that DWORD accesses to the expansion ROM must be supported.

The information in the ROMs is laid out to be compatible with existing Intel x86 Expansion ROM headers for ISA, EISA, and MC adapters, but it will also support other machine architectures. The information available in the header has been extended so that more optimum use can be made of the function provided by the adapter and so that the minimum amount of Memory Space is used by the runtime portion of the expansion ROM code.

The PCI Expansion ROM header information supports the following functions:

- A length code is provided to identify the total contiguous address space needed by the PCI device ROM image at initialization.
- An indicator identifies the type of executable or interpretive code that exists in the ROM address space in each ROM image.
- A revision level for the code and data on the ROM is provided.
- The Vendor ID and Device ID of the supported PCI device are included in the ROM.

One major difference in the usage model between PCI expansion ROMs and standard ISA, EISA, and MC ROMs is that ***the ROM code is never executed in place. It is always copied from the ROM device to RAM and executed from RAM****. This enables dynamic sizing of the code (for initialization and runtime) and provides speed improvements when executing runtime code.*

2.3.2.1. PCI Expansion ROM Contents

PCI device expansion ROMs may contain code (executable or interpretive) for multiple processor architectures. This may be implemented in a single physical ROM which can contain as many code images as desired for different system and processor architectures as shown in the picture below. Each image must start on a 512-byte boundary and must contain the PCI expansion ROM header. The starting point of each image depends on the size of previous images. The last image in a ROM has a special encoding in the header to identify it as the last image.

Image 0

Image 1

.

.

.

Image N

PCI Expansion ROM Structure

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

2.3.2.1.1. PCI Expansion ROM Header Format

The information required in each ROM image is split into two different areas. One area, the ROM header, is required to be located at the beginning of the ROM image. The second area, the PCI Data Structure, must be located in the first 64 KB of the image. The format for the PCI Expansion ROM header is given below. The offset is a hexadecimal number from the beginning of the image and the length of each field is given in bytes. Extensions to the PCI Expansion ROM Header and/or the PCI Data Structure may be defined by specific system architectures. Extensions for PC-AT compatible systems are described later.

Offset	Length	Value	Description
0h	1	55h	ROM Signature, byte 1
1h	1	AAh	ROM Signature, byte 2
2h-17h	16h	xx	Reserved (processor architecture unique data)
18h-19h	2	xx	Pointer to PCI Data Structure

ROM Signature The ROM Signature is a two-byte field containing a 55h in the first byte and AAh in the second byte. This signature must be the first two bytes of the ROM address space for each image of the ROM.

Pointer to PCI Data Structure The Pointer to the PCI Data Structure is a two-byte pointer in little endian format that points to the PCI Data Structure. The reference point for this pointer is the beginning of the ROM image.

2.3.2.1.2. PCI Data Structure Format

The PCI Data Structure must be located within the first 64 KB of the ROM image and must be DWORD aligned. The PCI Data Structure contains the following information:

Offset	Length	Description
0	4	Signature, the string "PCIR"
4	2	Vendor Identification
6	2	Device Identification
8	2	Pointer to Vital Product Data
A	2	PCI Data Structure Length
C	1	PCI Data Structure Revision
D	3	Class Code
10	2	Image Length
12	2	Revision Level of Code/Data
14	1	Code Type
15	1	Indicator
16	2	Reserved

Signature These four bytes provide a unique signature for the PCI Data Structure. The string "PCIR" is the signature with "P" being at offset 0, "C" at offset 1, etc.

Vendor Identification The Vendor Identification field is a 16-bit field with the same definition as the Vendor Identification field in the Configuration Space for this device.

Device Identification The Device Identification field is a 16-bit field with the same definition as the Device Identification field in the Configuration Space for this device.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

Pointer to Vital Product Data	The Pointer to Vital Product Data (VPD) is a 16-bit field that is the offset from the start of the ROM image and points to the VPD. This field is in little-endian format. The VPD must be within the first 64 KB of the ROM image. A value of 0 indicates that no Vital Product Data is in the ROM image. Section 6.4 describes the format and information contained in Vital Product Data.
PCI Data Structure Length	The PCI Data Structure Length is a 16-bit field that defines the length of the data structure from the start of the data structure (the first byte of the Signature field). This field is in little-endian format and is in units of bytes.
PCI Data Structure Revision	The PCI Data Structure Revision field is an eight-bit field that identifies the data structure revision level. This revision level is 0.
Class Code	The Class Code field is a 24-bit field with the same fields and definition as the class code field in the Configuration Space for this device.
Image Length	The Image Length field is a two-byte field that represents the length of the image. This field is in little-endian format, and the value is in units of 512 bytes.
Revision Level	The Revision Level field is a two-byte field that contains the revision level of the code in the ROM image.

Code Type The Code Type field is a one-byte field that identifies the type of code contained in this section of the ROM. The code may be executable binary for a specific processor and system architecture or interpretive code. The following code types are assigned:

Type	Description
0	Intel x86, PC-AT compatible
1	Open Firmware standard for PCI42
2-FF	Reserved

Indicator Bit 7 in this field tells whether or not this is the last image in the ROM. A value of 1 indicates "last image;" a value of 0 indicates that another image follows. Bits 0-6 are reserved.

2.3.2.2. Power-on Self Test (POST) Code

For the most part, system POST code treats add-in PCI devices identically to those that are soldered on to the motherboard. The one exception is the handling of expansion ROMs. POST code detects the presence of an option ROM in two steps. First the code determines if the device has implemented an Expansion ROM Base Address register in Configuration Space. If the register is implemented, the POST must map and enable the ROM in an unused portion of the address space, and check the first two bytes for the AA55h signature. If that signature is found, there is a ROM present; otherwise, no ROM is attached to the device.

If a ROM is attached, POST must search the ROM for an image that has the proper code type and whose Vendor ID and Device ID fields match the corresponding fields in the device.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

After finding the proper image, POST copies the appropriate amount of data into RAM. Then the device's initialization code is executed. Determining the appropriate amount of data to copy and how to execute the device's initialization code will depend on the code type for the field.

2.3.2.3. PC-compatible Expansion ROMs

This section describes further requirements on ROM images and the handling of ROM images that are used in PC-compatible systems. This applies to any image that specifies Intel x86, PC-AT compatible in the Code Type field of the PCI Data Structure, and any platform that is PC-compatible.

The standard header for PCI Expansion ROM images is expanded slightly for PC compatibility. Two fields are added, one at offset 02h provides the initialization size for the image. Offset 03h is the entry point for the expansion ROM INIT function.

Offset	Length	Value	Description
0h	1	55h	ROM Signature byte 1
1h	1	AAh	ROM Signature byte 2
2h	1	xx	Initialization Size - size of the code in units of 512 bytes.
3h	3	xx	Entry point for INIT function. POST does a FAR CALL to this location.
6h-17h	12h	xx	Reserved (application unique data)
18h-19h	2	xx	Pointer to PCI Data Structure

2.3.2.3.1. POST Code Extensions

POST code in these systems copies the number of bytes specified by the Initialization Size field into RAM, and then calls the INIT function whose entry point is at offset 03h. POST code is required to leave the RAM area where the expansion ROM code was copied to as writable until after the INIT function has returned. This allows the INIT code to store some static data in the RAM area, and to adjust the runtime size of the code so that it consumes less space while the system is running.

The PC-compatible specific set of steps for the system POST code when handling each expansion ROM are:

1. Map and enable the expansion ROM to an unoccupied area of the memory address space.
2. Find the proper image in the ROM and copy it from ROM into the compatibility area of RAM (typically 0C0000h to 0E0000h) using the number of bytes specified by Initialization Size.
3. Disable the Expansion ROM Base Address register.
4. Leave the RAM area writable and call the INIT function.
5. Use the byte at offset 02h (which may have been modified) to determine how much memory is used at runtime.

Before system boot, the POST code must make the RAM area containing expansion ROM code read-only. POST code must handle VGA devices with expansion ROMs in a special way. The VGA device's expansion BIOS must be copied to 0C0000h. VGA devices can be identified by examining the Class Code field in the device's Configuration Space.

2.3.2.3.2. INIT Function Extensions

PC-compatible expansion ROMs contain an INIT function that is responsible for initializing the I/O device and preparing for runtime operation. INIT functions in PCI expansion ROMs are allowed some

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

extended capabilities because the RAM area where the code is located is left writable while the INIT function executes.

The INIT function can store static parameters inside its RAM area during the INIT function. This data can then be used by the runtime BIOS or device drivers. This area of RAM will not be writable during runtime.

The INIT function can also adjust the amount of RAM that it consumes during runtime. This is done by modifying the size byte at offset 02h in the image. This helps conserve the limited memory resource in the expansion ROM area (0C0000h - 0DFFFFh).

For example, a device expansion ROM may require 24 KB for its initialization and runtime code, but only 8 KB for the runtime code. The image in the ROM will show a size of 24 KB, so that the POST code copies the whole thing into RAM. Then when the INIT function is running, it can adjust the size byte down to 8 KB. When the INIT function returns, the POST code sees that the runtime size is 8 KB and can copy the next expansion BIOS to the optimum location.

The INIT function is responsible for guaranteeing that the checksum across the size of the image is correct. If the INIT function modifies the RAM area in any way, then a new checksum must be calculated and stored in the image.

If the INIT function wants to completely remove itself from the expansion ROM area, it does so by writing a zero to the Initialization Size field (the byte at offset 02h). In this case, no checksum has to be generated (since there is no length to checksum across). On entry, the INIT function is passed three parameters: the bus number, device number, and function number of the device that supplied the expansion ROM. These parameters can be used to access the device being initialized. They are passed in x86 registers, [AH] contains the bus number, the upper five bits of [AL] contain the device number, and the lower three bits of [AL] contain the function number.

Prior to calling the INIT function, the POST code will allocate resources to the device (via the Base Address and Interrupt Line registers) and will complete any User Definable Features handling.

2.3.2.3.3. Image Structure

A PC-compatible image has three lengths associated with it, a runtime length, an initialization length, and an image length. The image length is the total length of the image and it must be greater than or equal to the initialization length.

The initialization length specifies the amount of the image that contains both the initialization and runtime code. This is the amount of data that POST code will copy into RAM before executing the initialization routine. Initialization length must be greater than or equal to runtime length. The initialization data that is copied into RAM must checksum to 0 (using the standard algorithm).

The runtime length specifies the amount of the image that contains the runtime code. This is the amount of data the POST code will leave in RAM while the system is operating. Again, this amount of the image must checksum to 0.

The PCI Data structure must be contained within the runtime portion of the image (if there is any) otherwise it must be contained within the initialization portion.

2.4. PCI PnP Expansion ROM Peculiarity

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

It is very clear from section 2.2 and 2.3 above that PCI specification and PnP BIOS specification has a "flaw" that can be exploited for our own purpose.

Both of the specification ***don't impose*** *that a PCI expansion ROM functionality has* ***to be cross-checked*** *by the system BIOS against the physical Class Code that is hardwired inside the PCI chip itself. Meaning, any PCI expansion card that implement an expansion ROM can be given a different functionality in its expansion ROM code, i.e. a functionality not related to the corresponding PCI chip itself. To be able to use PCI expansion ROM, the PCI chip only need to enable it's expansion ROM support in its XROMBAR.*

For example, we can hack a PCI SCSI controller card that has an expansion ROM to behave as if it's a LAN card from the system BIOS point of view. We will be able to "Boot from LAN" by using this card.

We have been experimenting with this "flaw" and it works as predicted above. By making the PCI expansion ROM contents to conform to an RPL(Remote Program Load) PCI card (LAN card that supports boot from LAN), we were able to execute our custom made PCI expansion ROM code. The detail of PCI card that we have tested as follows :

- Realtek 8139A LAN Card (Vendor ID = 10EC, Device ID = 8139). This is a real PCI LAN card, used for comparison purposes. We equipped it with an Atmel AT29C512 flash rom (64 KB), which is purchased separately since the card doesn't come with any flash rom at all. The custom PCI expansion ROM were flashed using flash program provided by Realtek (rtflash.exe). We have enabled and set the address space consumed by the flash rom chip in XROMBAR of the Realtek chip with Realtek's rset8139.exe program prior to flashing the custom made expansion ROM. Keep in mind that the expansion ROM chip **is not accessible** until the XROMBAR has been initialized with the right value, unless the XROMBAR value has been hardwired to unconditionally support certain address space for expansion ROM chip.
- Adaptec AHA-2940U SCSI controller card (Vendor ID = 9004, Device ID = 8178). It has been equipped with a soldered PLCC SST 29EE512 flash rom (64 KB). The custom PCI expansion ROM code flashed using flash program (flash4.exe) from Adaptec. This utility is distrbuted along with adaptec PCI SCSI controller BIOS update. The SCSI controller chip has its XROMBAR value hardwired to support 64 KB flash rom chip. The result is a bit weird, no matter how we changed the BIOS setup (boot from LAN option), the PCI initialization routine (not the BEV routine) always get called. We think this is due to the controller's chip Subclass Code and Interface Code inside the PCI chip (SCSI bus controller boot device). The hacked card behave as if it's a real PCI LAN Card, i.e. the system boots from the hacked card if we set the mainboard BIOS to boot from LAN and our experimental BEV routine inside the custom PCI expansion ROM code is invoked.

3. Implementation Sample

This section provides an implementation sample that has been tested in our test bed. The sample is a custom PCI expansion ROM that will be executed after the main board BIOS has done initialization. The sample is "jumped into" through its BEV by the main board BIOS during bootstrap.

3.1. Hardware

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

The hardware used for this sample is Adaptec AHA-2940U PCI SCSI controller card (Vendor ID = 9004, Device ID = 8178). It has a soldered PLCC SST 29EE512 flash rom (64 KB) for its firmware. It cost Rp. 25.000 (around US$2.5). We get this hardware from refurbished PC component seller. The PC that's used for expansion ROM development and also as the target platform has the following hardware configuration :

Processor	**:** Intel Celeron 300A, overclocked to 518 MHz by using ABIT Slotket II adapter
Mainboard	**:** Iwill VD133 (slot 1) with VIA693A northbridge and VIA596B southbridge
Videocard	**:** PowerColor Nvidia Riva TNT2 M64 32MB
RAM	**:** 256MB SDRAM with unknown chip
Soundcard	**:** Addonics Yamaha YMF724
Network Card	**:** Realtek RTL8139C
"Hacked" PCI Card	**:** Adaptec AHA-2940U PCI SCSI controller card
Harddrive	**:** Maxtor 20GB 5400RPM
CDROM	**:** Teac 40X
Monitor	**:** Samsung SyncMaster 551v (15')

3.2. Software Development Tool

There are three kind of software that are needed for the development of this sample :

1. Development environment that provides compiler, assembler and linker for x86. We are using GNU Software, i.e. GNU AS assembler, GNU LD linker, GNU GCC compiler, and GNU Make. These development tool were running on Slackware Linux 9.0 in our development PC. We are using Vi as the editor and Bash shell to run these tools. Note that the GNU LD linker that's used for development must support ELF object file format to be able to compile our sample source code (provided in later section). Generally all Linux distribution support this object file format by default. As an addition, we are using hexdump utility in linux to inspect the result of our development.
2. PCI PnP expansion ROM checksum patcher. As we see in section 2, a valid PCI expansion ROM has a lot of checksums value that need to be fulfilled. Since our development environment can not provide us with that, we develop our own custom tool for it. The source code of this tool is provided in later section.
3. Adaptec PCI expansion ROM flash utility for AHA-2940UW. The utility is named flash4.exe, it comes with the Adaptec AHA-2940UW BIOS version 2.57.2 distribution. It's used to flash our custom made expansion ROM code into the flash rom of the card. We are using bootable cdrom to get into realmode dos and invoke the flash utility, it also needs DOS4GW that's provided with the adaptec PCI BIOS distribution.

3.3. The PCI PnP Expansion ROM Source Code

The basic run-down of what happens when the compiled source code executed as follows:

1. During POST, the system bios (original.tmp in Award BIOS) look for implemented PCI expansion ROMs from every PCI expansion card by testing XROMBAR (Expansion ROM Base Address Register) of each card. If it is implemented (XROMBAR consumed address space), then system BIOS will copy the PCI expansion ROM from the address pointed to by the XROMBAR (ROM) to RAM in the expansion ROM area (C0000h - DFFFFh physical

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

address). Then system bios will jump to the init function of the pci expansion rom. After the pci expansion rom has done its initialization, execution is back to system bios. System bios will check the runtime size of the pci expansion rom that has been initialized previously, it will copy the next pci expansion rom from another PCI card (if exist) to RAM at address _previous_expansion_rom_address_+_its_runtime_size_. This effectively "trashed" unneeded portion of the previous expansion rom.

2. Having done all PCI expansion ROM initialization, system BIOS will write-protect the expansion rom area in RAM (C0000h - DFFFFh physical address). We haven't carry further experiment to prove this. We will just protect our code against this possiblity by copying ourself to 0000:0000h in RAM.
3. System BIOS then do bootstrap. It looks for IPL (Initial Program Loader) device, if we set up the main bios to boot from LAN as default, the IPL device will be our "LAN Card". Int 19h (bootstrap) will point into the PnP option rom BEV of the "LAN card" and passes execution into our code there. So we're executing code in the write-protected RAM pointed to by the BEV. Unless we're loading part of this code into RAM area that's read-write enabled and execute from there, there's no writeable area in our code.
4. The custom PCI PnP expansion ROM code then executed. The expansion ROM code then copies itself from the expansion ROM area in RAM (inside C_0000h - D_FFFFh region) to physical address 0000_0000h and continue execution from there. After copying itself, then the code switches the machine into 32-bit protected mode and displays "Hello World" in the display. Then the code enters an infinite loop.

In the next two sections we will be dealing with the expansion rom source code. The first section will provide the source code of the expansion ROM itself, while the second one will provide the source code of the utility used to patch the binary file resulting from the first-section's source code into a valid PCI PnP Expansion ROM.

3.3.1. Core PCI PnP Expansion ROM Source Code

The purpose of the source code that is provided in this section is to show how a PCI PnP Expansion ROM source code might look like. The role of each file as follows :

- Makefile : makefile used to build the expansion ROM binary
- crt0.S : assembly language file that contains all the headers needed, entry point for the BEV. After done with initialization task, it swithches the machine to 32-bit protected mode.
- main.c : c language source code that is jumped after crt0.S executed. It displays the "Hello World" message then enters infinite loop.
- pci_rom.ld : linker script used to perform linking and relocation to the object file resulting from crt0.S and main.c.

```
# -----------------------------------------------------------------------
# Copyright (C) Darmawan Mappatutu Salihun
# File name : Makefile
# This file is released to the public for non-commercial use only
# -----------------------------------------------------------------------

CC= gcc
CFLAGS= -c
LD= ld
LDFLAGS= -T pci_rom.ld

ASM= as

OBJCOPY= objcopy
OBJCOPY_FLAGS= -v -O binary
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
OBJS:= crt0.o main.o
ROM_OBJ= rom.elf
ROM_BIN= rom.bin
ROM_SIZE= 65536

all: $(OBJS)
        $(LD) $(LDFLAGS) -o $(ROM_OBJ) $(OBJS)
        $(OBJCOPY) $(OBJCOPY_FLAGS) $(ROM_OBJ) $(ROM_BIN)

        build_rom $(ROM_BIN) $(ROM_SIZE)

crt0.o: crt0.S
        $(ASM) -o $@  $<

%.o: %.c
        $(CC) -o $@  $(CFLAGS) $<

clean:
        rm -rf *~ *.o *.elf *.bin
```

```
# -------------------------------------------------------------------------------
-------------------------------
# Copyright (C)  Darmawan Mappatutu Salihun
# File name : crt0.S
# This file is released to the public for non-commercial use only
# -------------------------------------------------------------------------------
--------------------------------

.text
.code16 # Real mode by default (prefix 66 or 67 to 32 bits instructions)

# ------------------------- WARNING!!! -----------------------------------------
# 1. Make sure to synchronize the absolute address used to load the OS code here
and
#    in the address defined in the linker script
# 2. Make sure the rom size is correct
#

rom_size        = 0x04      # ROM size in multiple of 512 bytes
os_load_seg     = 0x0000    # this is working if lgdt is passed with an absolute
address
os_code_size    = ((rom_size - 1)*512)
os_code_size16  = ( os_code_size / 2 )

# -------------------------------------------
#       Option rom header
#
    .word 0xAA55   # Rom signature
    .byte rom_size # Size of this ROM, see definition above
    jmp _init      # jump to PCI initialization function

    .org 0x18
    .word _pci_data_struct # Pointer to PCI HDR structure (at 18h)
    .word _pnp_header      # PnP Expansion Header Pointer (at 1Ah)

# --------------------------------------------
# PCI data structure
#
_pci_data_struct:
    .ascii  "PCIR"   # PCI Header Sign
    .word   0x9004   # Vendor ID
    .word   0x8178   # Device ID
    .word   0x00     # VPD
    .word   0x18     # PCI data struc length (byte)
    .byte   0x00     # PCI Data struct Rev
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
    .byte   0x02     # Base class code, 02h == Network Controller
    .byte   0x00     # Sub class code = 00h and interface = 00h -->Ethernet
Controller
    .byte   0x00     # Interface code, see PCI Rev2.1 Spec Appendix D
    .word   rom_size # Image length in mul of 512 byte, little endian format
    .word   0x00     # rev level
    .byte   0x00     # Code type = x86
    .byte   0x80     # last image indicator
    .word   0x00     # reserved

# ----------------------------
# PnP ROM Bios Header
#
_pnp_header:
    .ascii "$PnP"          # PnP Rom header sign
    .byte  0x01            # Structure Revision
    .byte  0x02            # Header structure Length in mul of 16 bytes
    .word  0x00            # Offset to next header (00 if none)
    .byte  0x00            # reserved
    .byte  0x00            # 8-Bit checksum for this header, calculated and patched
by build_rom
    .long  0x00            # PnP Device ID (0h in Realtek RPL ROM, we just follow
it)
    .word  0x00            # pointer to manufacturer string, we use empty string
    .word  0x00            # pointer to product string, we use empty string
    .byte  0x02,0x00,0x00 # Device Type code 3 byte
    .byte  0x14            # Device Indicator, 14h from Realtek RPL ROM-->See Page
18 of
                           # PnP BIOS spec., Lo nibble (4) means IPL device

    .word  0x00     # Boot Connection Vector, 00h = disabled
    .word  0x00     # Disconnect Vector, 00h = disabled
    .word  _start   # Bootstrap Entry Vector (BEV)
    .word  0x00     # reserved
    .word  0x00     # Static resource Information vector (0000h if unused)

#--------------------------------------------------------------------
# PCI Option ROM initialization Code (init function)
#
_init:
  andw $0xCF, %ax # inform system BIOS that an IPL device attached
  orw  $0x20, %ax # see PnP spec 1.0A p21 for info's

  lret # return far to system BIOS

#--------------------------------------------------------------------
# entry point/BEV implementation (invoked during bootstrap / int 19h)
#
  .global _start # entry point

_start:
  movw $0x9000, %ax # setup temporary stack
  movw %ax, %ss     # ss = 0x9000

# move ourself from "ROM" -> RAM 0x0000
  movw %cs, %ax          # initialize source address
  movw %ax, %ds
  movw $os_load_seg, %ax # point to OS segment
  movw %ax, %es
  movl $os_code_size16, %ecx
  subw %di, %di
  subw %si, %si
  cld
  rep
  movsw
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
  ljmp $os_load_seg, $_setup

_setup:
  movw %cs, %ax  # initialize segment registers (this is needed since lgdt is %ds
dependent??)
  movw %ax, %ds

enable_a20:
        cli

        call    a20wait
        movb    $0xAD, %al
        outb    %al, $0x64

        call    a20wait
        movb    $0xD0, %al
        outb    %al, $0x64

        call    a20wait2
        inb     $0x60, %al
        pushl   %eax

        call    a20wait
        movb    $0xD1, %al
        outb    %al, $0x64

        call    a20wait
        popl    %eax
        or      $2, %al
        outb    %al, $0x60

        call    a20wait
        movb    $0xAE, %al
        outb    %al, $0x64

        call    a20wait
                 jmp      continue

a20wait:
1:    movl    $65536, %ecx
2:    inb     $0x64, %al
      test    $2, %al
      jz      3f
      loop    2b
      jmp     1b
3:    ret


a20wait2:
1:    movl    $65536, %ecx
2:    inb     $0x64, %al
      test    $1, %al
      jnz     3f
      loop    2b
      jmp     1b
3:    ret

continue:
    sti # enable interrupt

# ----------------------------------------------------------------------
# Switch to P-Mode and jump to C-Compiled kernel
#
  cli            # disable interrupt

  lgdt gdt_desc  # load GDT to GDTR (we load both limit and base address)
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
  movl %cr0, %eax     # switch to P-Mode
  or   $1, %eax
  movl %eax, %cr0     # haven't yet in P-Mode, we need a FAR Jump

  .byte 0x66, 0xea   # prefix + jmpi-opcode (this force P-Mode to be reached i.e.
CS updated)
  .long do_pm        # 32-bit linear address (jump target)
  .word SEG_CODE_SEL # code segment selector

.code32
do_pm:
  xorl %esi, %esi
  xorl %edi, %edi
  movw $0x10, %ax  # Save data segment identifier (see GDT)
  movw %ax, %ds
  movw $0x18, %ax  # Save stack segment identifier
  movw %ax, %ss
  movl $0x90000, %esp

  jmp  main # jump to main function

  .align 8, 0 # align GDT in 8 bytes boundary

# -----------------------------------------------------
#                        GDT definition
#
gdt_marker:   # dummy Segment Descriptor (GDT)
  .long 0
  .long 0

SEG_CODE_SEL = ( . - gdt_marker)
SegDesc1:      # kernel CS (08h) PL0, 08h is an identifier
  .word 0xffff # seg_length0_15
  .word 0      # base_addr0_15
  .byte 0      # base_addr16_23
  .byte 0x9A   # flags
  .byte 0xcf   # access
  .byte 0      # base_addr24_31

SEG_DATA_SEL = ( . - gdt_marker)
SegDesc2:      # kernel DS (10h) PL0
  .word  0xffff # seg_length0_15
  .word  0      # base_addr0_15
  .byte 0      # base_addr16_23
  .byte 0x92   # flags
  .byte 0xcf   # access
  .byte 0      # base_addr24_31

SEG_STACK_SEL = ( . - gdt_marker)
SegDesc3:      # kernel SS (18h) PL0
  .word 0xffff # seg_length0_15
  .word 0      # base_addr0_15
  .byte 0      # base_addr16_23
  .byte 0x92   # flags
  .byte 0xcf   # access
  .byte 0      # base_addr24_31
gdt_end:

gdt_desc:  .word (gdt_end - gdt_marker - 1) # GDT limit
           .long gdt_marker # physical addr of GDT
```

```
/* ---------------------------------------------------------------------------------
--------------------------------
  Copyright (C)  Darmawan Mappatutu Salihun
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
  File name : main.c
  This file is released to the public for non-commercial use only
  ----------------------------------------------------------------------------------
--------------------------------*/

unsigned char in(unsigned short _port)
{
  // "=a" (result) means: put AL register in variable result when finished
  // "d" (_port) means: load EDX with _port
  unsigned char result;
  __asm__  ("in %%dx, %%al" : "=a" (result) : "d" (_port));
  return result;
}

void out(unsigned short _port, unsigned char _data)
{
  // "a" (_data) means: load EAX with _data
  // "d" (_port) means: load EDX with _port
  __asm__ ("out %%al, %%dx" : :"a" (_data), "d" (_port));
}

void clrscr()
{
  unsigned char *vidmem = (unsigned char *)0xB8000;
  const long size = 80*25;
  long loop;

  // Clear visible video memory
  for (loop=0; loop < size; loop++){
    *vidmem++ = 0;
    *vidmem++ = 0xF;
  }

  // Set cursor position to 0,0
  out(0x3D4, 14);
  out(0x3D5, 0);
  out(0x3D4, 15);
  out(0x3D5, 0);

}


void print(const char *_message)
{
  unsigned short offset;
  unsigned long i;
  unsigned char *vidmem = (unsigned char *)0xB8000;

  // Read cursor position
  out(0x3D4, 14);
  offset = in(0x3D5) << 8;
  out(0x3D4, 15);
  offset |= in(0x3D5);

  // Start at writing at cursor position
  vidmem += offset*2;

  // Continue until we reach null character
  i = 0;
  while (_message[i] != 0) {
    *vidmem = _message[i++];
        vidmem += 2;
  }

  // Set new cursor position
  offset += i;
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
  out(0x3D5, (unsigned char)(offset));
  out(0x3D4, 14);
  out(0x3D5, (unsigned char)(offset >> 8));
}


int main()
{
  const char *hello = "Hello World";
  clrscr();
  print(hello);

  for(;;);

  return 0;
}
```

```
/*=========================================================================*/
/* Copyright (C) Darmawan Mappatutu Salihun                                */
/* File name : pci_rom.ld                                                  */
/* This file is released to the public for non-commercial use only         */
/*=========================================================================*/

OUTPUT_FORMAT("elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)

__boot_vect = 0x0000;

SECTIONS
{

    .text __boot_vect :
    {
      *( .text)
    } = 0x00

    .rodata ALIGN(4) :
    {
      *( .rodata)
    } = 0x00

    .data ALIGN(4) :
    {
      *( .data)
    } = 0x00

    .bss ALIGN(4) :
    {
      *( .bss)
    } = 0x00

}
```

3.3.2. PCI PnP Expansion ROM Checksum Utility Source Code

The source code that is provided in this section is used to build the **build_rom** utility which is used to patch the checksums of the PCI PnP Expansion ROM binary produced by section 3.3.2. The role of each file as follows :

- Makefile : makefile used to build the utility.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

- build_rom.c : c language source code for the **build_rom** utility.

```
# ---------------------------------------------------------------------------
# Copyright (C) Darmawan Mappatutu Salihun
# File name : Makefile
# This file is released to the public for non-commercial use only
# ---------------------------------------------------------------------------


CC= gcc
CFLAGS= -Wall -O2 -march=i686 -mcpu=i686 -c
LD= gcc
LDFLAGS=

all: build_rom.o
	$(LD) $(LDFLAGS) -o build_rom build_rom.o

	cp build_rom ../

%.o: %.c
	$(CC) $(CFLAGS) -o $@ $<


clean:
	rm -rf *~ build_rom *.o
```

```
/* ------------------------------------------------------------------------------------
------

 Copyright (c) Darmawan MS
 File name : build_rom.c
 This file is released to the public for non-commercial use only

 Description :

 This program zero-extend its input binary file and then patch it
 into a valid PCI PnP ROM binary.
 ------------------------------------------------------------------------------------
------ */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef unsigned char     u8;
typedef unsigned short    u16;
typedef unsigned int      u32;

enum {
MAX_FILE_NAME             = 100,

ITEM_COUNT                        =       1,
ROM_SIZE_INDEX            = 0x2,
PnP_HDR_PTR                       = 0x1A,
PnP_CHKSUM_INDEX = 0x9,
PnP_HDR_SIZE_INDEX        = 0x5,
ROM_CHKSUM                        = 0x10, /* reserved position in PCI PnP ROM, that
can be used */
};


static int
ZeroExtend(char * f_name, u32 target_size)
{
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
  FILE* f_in;
  long file_size, target_file_size, padding_size;
  char* pch_buff;

  target_file_size = target_size; // cast ulong to long

  if( (f_in = fopen(f_name, "ab")) == NULL)
  {
        printf("error opening file\n closing program...\n");
        return -1;
  }

  if(fseek(f_in, 0, SEEK_END) != 0)
  {
        printf("error seeking file\n closing program...\n");
        fclose(f_in);
        return -1;
  }

  if( (file_size = ftell(f_in)) == -1)
  {
        printf("error counting file size\n closing program...\n");
        fclose(f_in);
        return -1;
  }

  if( file_size >= target_file_size)
  {
        printf("Input error, Target file size is smaller than the original file
size\n");
        fclose(f_in);
    return -1;
  }

  /*
    Zero extend the target file
  */
  padding_size = target_file_size - file_size;

  pch_buff = (char*) malloc(sizeof(char) * padding_size );

  if(NULL != pch_buff) {
        memset(pch_buff, 0, sizeof(char) * padding_size );
        fseek(f_in, 0, SEEK_END);
        fwrite( pch_buff, sizeof(char), padding_size, f_in);
        fclose(f_in);
        free(pch_buff);
        return 0;//success

  } else {
        fclose(f_in);
        return -1;

  }

}

static u8 CalcChecksum(FILE* fp, u32 size)
{
u32  position = 0x00;/* Position of file pointer */
u8 checksum = 0x00;

        /* set file pointer to the beginning of file */
        if(!fseek(fp,0,SEEK_SET))
        {
                /*
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
                calculate 8 bit checksum 8
                file size = size * 512 byte = size * 0x200
                */

                for(; position < (size * 0x200) ; position++)
                {
                        checksum = ( (checksum + fgetc(fp)) % 0x100);
                }

                printf("calculated checksum = %#x \n",checksum);

        }

        else
        {
        printf("function CalcChecksum:Failed to seek through the beginning of
file\n");
        }


        return checksum;

}

static int
Patch2PnpRom(char* f_name)
{
        FILE*   fp;
        u8      checksum_byte;
        u32     rom_size; /* size of ROM source code in multiple of 512 bytes */
        u8      pnp_header_pos;
        u8      pnp_checksum = 0x00;
        u8      pnp_checksum_byte;
        u8      pnp_hdr_counter = 0x00;
        u8      pnp_hdr_size;

        if( (fp = fopen( f_name , "rb+")) == NULL)
        {
                printf("Error opening file\nclosing program ...");
                return -1;
        }


        /* Save ROM source code file size which is located
        at index 0x2 from beginning of file (zero based index) */

        fseek(fp, ROM_SIZE_INDEX, SEEK_SET);
        rom_size = fgetc(fp);


                /* Patch the PnP Header checksum */
                if(fseek(fp,PnP_HDR_PTR,SEEK_SET) != 0)
                {
                        printf("Error seeking PnP Header");
                        fclose(fp);
                        return -1;
                }

                pnp_header_pos = fgetc(fp);/* save PnP header offset */

                if(fseek(fp,(pnp_header_pos + PnP_HDR_SIZE_INDEX), SEEK_SET) != 0)
                {
                        printf("Error seeking PnP Header Checksum\n");
                        fclose(fp);
                        return -1;
                }
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
                pnp_hdr_size = fgetc(fp);/* save PnP header size*/

                /* reset current checksum to 0x00 so that
                the checksum won't be wrong if calculated */

                if(fseek(fp,(pnp_header_pos + PnP_CHKSUM_INDEX),SEEK_SET) != 0)
                {
                        printf("Error seeking PnP Header Checksum\n");
                        fclose(fp);
                        return -1;
                }

                if(fputc(0x00,fp) == EOF)
                {
                        printf("Error resetting PnP Header checksum value\n");
                        fclose(fp);
                        return -1;
                }

                /* calculate PnP Header Checksum */
                if(fseek(fp,pnp_header_pos,SEEK_SET) != 0)
                {
                        printf("Error seeking to calculate PnP Header checksum");
                        fclose(fp);
                        return -1;
                }

                        /*
                        PnP BIOS Header size is calculated in every 16 bytes
increment
                        */
                        for(; pnp_hdr_counter < (pnp_hdr_size * 0x10) ;
pnp_hdr_counter++)
                        {
                                pnp_checksum = ( (pnp_checksum + fgetc(fp)) %
0x100);
                        }

                        if(pnp_checksum != 0 ) {
                                pnp_checksum_byte = 0x100 - pnp_checksum;
                        } else {
                                pnp_checksum_byte = 0;
                        }

                /* write PnP Header Checksum */
                fseek(fp,(pnp_header_pos + PnP_CHKSUM_INDEX), SEEK_SET);
                fputc(pnp_checksum_byte ,fp);



        /* Overall file checksum handled from here on */

        /* reset current checksum on checksum byte */
        if(     fseek(fp, ROM_CHKSUM, SEEK_SET) != 0 ) {
                fclose(fp);
                return -1;
        } else {
                fputc(0x00,fp);
        }


        /* calculate checksum byte */
        if(CalcChecksum(fp,rom_size) == 0x00) {
                checksum_byte = 0x00; /* checksum already O.K */
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
        } else {
                checksum_byte = 0x100 - CalcChecksum(fp,rom_size);
        }


        /* Write Checksum byte */

                /* Put the file pointer at the checksum byte */
                if(fseek(fp, ROM_CHKSUM, SEEK_SET) != 0)
                {
                        printf("Failed to seek through the file\nclosing program
...");
                        fclose(fp);
                        return -1;
                } else {
                        /* write the checksum to the checksum byte in the file */
                        fputc(checksum_byte, fp);
                }


        /* write to disk */
        fclose(fp);

        printf("PnP ROM successfully created\n");

        return 0;

}


int main(int argc, char* argv[])
{
        char out_f_name[MAX_FILE_NAME];
        u32  target_size;
        char* pch_temp[15];

        if(argc != 3) /* not enough parameter */
        {
                printf("Usage: %s  [input_filename]
[target_binary_size]\n",argv[0]);
                printf("input_filename = binary file that need to be patched into
PCI PnP ROM\n"
                            "target_binary_size = the intended size of the PCI PnP
ROM\n");
                return -1;
        }

        strncpy(out_f_name, argv[1], MAX_FILE_NAME - 1);

        target_size = strtoul(argv[2], pch_temp, 10);
        if( 0 != (target_size % 512) ) {
                printf("Error on input parameter. Invalid target binary size!\n");
                return -1;
        }


        /* argv[1] is pointer to the filename parameter from user */
        if(ZeroExtend(out_f_name, target_size) != 0)
        {
                printf("Error zero-extending output file ! \nclosing program ...");
                return -1;
        }

        if(Patch2PnpRom(out_f_name) != 0)
        {
                printf("Error patching checksums ! \nclosing program ...");
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
                return -1;
        }
        return 0;
}
```

3.3.3. PCI PnP Expansion ROM Build Step

The steps below is needed to be carried out to build a valid PCI PnP Expansion ROM from the code provided above. We are assuming that all of the command mentioned here is typed in a bash shell within Linux. We are using Slackware 9.0 linux distribution in our development testbed.

1. Create a new directory for the Core PCI expansion ROM source code. From now on we will regard this directory as the **root directory**.
2. Copy all of the core source code files into the root directory.
3. Create a new directory inside the root directory. From now on we will regard this directory as the **rom_tool directory**.
4. Copy all of the PCI PnP Expansion ROM checksum utility source code files into the root directory.
5. Invoke "make" from within rom_tool directory. This will build the utility needed for later step. The resulting build_rom utility will be copied automatically to root directory, where it will be needed in later build step.
6. Invoke "make" from within root directory. This will build the valid PCI PnP expansion rom that can be directly flashed to target PCI card (the "hacked" Adaptec AHA 2940 card). This expansion rom binary will be named **rom.bin**.

The result of these build steps is shown below. We are using hexdump utility from our Slackware Linux to obtain the result by invoking "hexdump -C rom.bin" in bash shell.

```
00000000  55 aa 04 eb 4f 00 00 00  00 00 00 00 00 00 00 00  |U...O...........|
00000010  2a 00 00 00 00 00 00 00  1c 00 34 00 50 43 49 52  |*.........4.PCIR|
00000020  04 90 78 81 00 00 18 00  00 02 00 00 04 00 00 00  |..x.............|
00000030  00 80 00 00 24 50 6e 50  01 02 00 00 00 5a 00 00  |....$PnP.....Z..|
00000040  00 00 00 00 00 00 02 00  00 14 00 00 00 00 5b 00  |..............[.|
00000050  00 00 00 00 25 cf 00 83  c8 20 cb b8 00 90 8e d0  |....%.... ......|
00000060  8c c8 8e d8 b8 00 00 8e  c0 66 b9 00 03 00 00 29  |.........f.....)|
00000070  ff 29 f6 fc f3 a5 ea 7b  00 00 00 8c c8 8e d8 fa  |.).....{........|
00000080  e8 2e 00 b0 ad e6 64 e8  27 00 b0 d0 e6 64 e8 31  |......d.'....d.1|
00000090  00 e4 60 66 50 e8 19 00  b0 d1 e6 64 e8 12 00 66  |..`fP......d...f|
000000a0  58 0c 02 e6 60 e8 09 00  b0 ae e6 64 e8 02 00 eb  |X...`......d....|
000000b0  22 66 b9 00 00 01 00 e4  64 a8 02 74 04 e2 f8 eb  |"f......d..t....|
```

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

```
000000c0  f0 c3 66 b9 00 00 01 00  e4 64 a8 01 75 04 e2 f8  |..f......d..u...|
000000d0  eb f0 c3 fb fa 0f 01 16  28 01 0f 20 c0 66 83 c8  |........(.. .f..|
000000e0  01 0f 22 c0 66 ea ec 00  00 00 08 00 31 f6 31 ff  |..".f.......1.1.|
000000f0  66 b8 10 00 8e d8 66 b8  18 00 8e d0 bc 00 00 09  |f.....f.........|
00000100  00 e9 ea 01 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000110  ff ff 00 00 00 9a cf 00  ff ff 00 00 00 92 cf 00  |................|
00000120  ff ff 00 00 00 92 cf 00  1f 00 08 01 00 00 00 00  |................|
00000130  55 89 e5 83 ec 04 8b 45  08 66 89 45 fe 0f b7 55  |U......E.f.E...U|
00000140  fe ec 88 45 fd 0f b6 45  fd c9 c3 55 89 e5 83 ec  |...E...E...U....|
00000150  04 8b 45 08 8b 55 0c 66  89 45 fe 88 55 fd 0f b6  |..E..U.f.E..U...|
00000160  45 fd 0f b7 55 fe ee c9  c3 55 89 e5 83 ec 18 c7  |E...U....U......|
00000170  45 fc 00 80 0b 00 c7 45  f8 d0 07 00 00 c7 45 f4  |E......E......E.|
00000180  00 00 00 00 8b 45 f4 3b  45 f8 7c 02 eb 1d 8b 45  |.....E.;E.|....E|
00000190  fc c6 00 00 8d 45 fc ff  00 8b 45 fc c6 00 0f 8d  |.....E....E.....|
000001a0  45 fc ff 00 8d 45 f4 ff  00 eb d9 c7 44 24 04 0e  |E....E......D$..|
000001b0  00 00 00 c7 04 24 d4 03  00 00 e8 8c ff ff ff c7  |.....$..........|
000001c0  44 24 04 00 00 00 00 c7  04 24 d5 03 00 00 e8 78  |D$.......$.....x|
000001d0  ff ff ff c7 44 24 04 0f  00 00 00 c7 04 24 d4 03  |....D$.......$..|
000001e0  00 00 e8 64 ff ff ff c7  44 24 04 00 00 00 00 c7  |...d....D$......|
000001f0  04 24 d5 03 00 00 e8 50  ff ff ff c9 c3 55 89 e5  |.$.....P.....U..|
00000200  83 ec 18 c7 45 f4 00 80  0b 00 c7 44 24 04 0e 00  |....E......D$...|
00000210  00 00 c7 04 24 d4 03 00  00 e8 2d ff ff ff c7 04  |....$.....-.....|
00000220  24 d5 03 00 00 e8 06 ff  ff ff 66 0f b6 c0 c1 e0  |$.........f.....|
00000230  08 66 89 45 fe c7 44 24  04 0f 00 00 00 c7 04 24  |.f.E..D$.......$|
00000240  d4 03 00 00 e8 02 ff ff  ff c7 04 24 d5 03 00 00  |...........$....|
00000250  e8 db fe ff ff 66 0f b6  d0 0f b7 45 fe 09 d0 66  |.....f.....E...f|
00000260  89 45 fe 0f b7 45 fe 8d  14 00 8d 45 f4 01 10 c7  |.E...E.....E....|
00000270  45 f8 00 00 00 00 8b 45  f8 03 45 08 80 38 00 75  |E......E..E..8.u|
00000280  02 eb 1b 8b 55 f4 8b 45  f8 03 45 08 0f b6 00 88  |....U..E..E.....|
00000290  02 8d 45 f8 ff 00 8d 45  f4 83 00 02 eb d8 0f b7  |..E....E........|
000002a0  55 fe 8b 45 f8 8d 04 10  66 89 45 fe 0f b6 45 fe  |U..E....f.E...E.|
000002b0  89 44 24 04 c7 04 24 d5  03 00 00 e8 8b fe ff ff  |.D$...$.........|
000002c0  c7 44 24 04 0e 00 00 00  c7 04 24 d4 03 00 00 e8  |.D$.......$.....|
000002d0  77 fe ff ff 0f b7 45 fe  c1 e8 08 0f b6 c0 89 44  |w.....E........D|
000002e0  24 04 c7 04 24 d5 03 00  00 e8 5d fe ff ff c9 c3  |$...$.....].....|
000002f0  55 89 e5 83 ec 08 83 e4  f0 b8 00 00 00 00 29 c4  |U.............).|
00000300  c7 45 fc 1c 03 00 00 e8  5d fe ff ff 8b 45 fc 89  |.E......]....E..|
00000310  04 24 e8 e6 fe ff ff eb  fe 00 00 00 48 65 6c 6c  |.$..........Hell|
00000320  6f 20 57 6f 72 6c 64 00  00 00 00 00 00 00 00 00  |o World.........|
00000330  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00010000
```

3.4. Testing The Custom Build PCI PnP Expansion ROM

Testing the binary is trivial. We used the aforementioned flash4.exe to flash our rom.bin file from real mode DOS by invoking the command below :

```
flash4.exe -w rom.bin
```

Then we can see the result by activating **boot from lan** in our BIOS. We will see the "Hello World" displayed on the screen.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

3.5. Potential Bug and Its Workaround

We have to emphasize that anyone who is building a PCI expansion ROM has to check the value of the Vendor ID and device ID within their source code. It's possible that the expansion ROM code is not executed at all (not "jumped-into" by the system bios) since there is a mismatch Vendor ID or Device ID between the expansion ROM and the value hardwired into the PCI chip. We haven't done further work on this issue, but we strongly suggest to avoid this mismatch.

There is a very specific circumstance where the PCI initialization routine that we make being screwed-up during development using this board (Adaptec AHA-2940U SCSI controller card with soldered PLCC SST 29EE512 flash rom). In this very specific case we were not be able to complete the boot of the testbed PC, since the mainboard BIOS possibly will hang at POST. In our case this was due to wrong placement of the entry point to the PCI initialization routine. This entry point is a jump instruction at offset 03h from the beginning of the rom binary image file, it should have been placed there but we inadvertently placed it at offset 04h. Thus, during the execution of PCI init function, the PC hangs. The "brute force" workaround for this is as follows :

1. Install the corresponding "screwed-up" SCSI controller card into one of the PCI slot, in case you haven't done it yet. Of course with the PC turned off.
2. Short circuit the lowest address pins of the soldered flash rom during boot until we can get into pure dos mode. In our case we use a metal wire for that. This wire is "installed" while the PC powered-off and unplugged from electrical source. We were short-circuiting address pin 0 (A0) and address pin 1 (A1). Short-circuiting A0 and A1 is enough, since we only need to generate a wrong PCI ROM header in the first 2 bytes. To know which of the pin is the lowest address pin, find the datasheet of the flash rom from it's manufacturer website. This step is done to "purposely generate checksum error" in the PCI ROM header "magic number", i.e. 55AAh. The reason behind this step is: if the PCI ROM header "magic number" is erratic, mainboard BIOS will ignore this PCI expansion rom bios. Thus, we can proceed to boot to DOS and going through POST without hang.
3. When we get into pure DOS, release the wire/conductor used to short-circuit the address pins. Therefore, we will be able to flash the correct rom binary into the flash rom chip of the SCSI controller flawlessly.
4. Flash the correct rom binary file to the flash rom chip. Then reboot to make sure everything is OK. The point is, if we are using a hacked SCSI controller card, the PCI init function has to be working flawlessly, since it's always executed by the mainboard BIOS on boot. We are not so sure about the reason, but it seems to be system BIOS checks the physical class code of the chip in it's PCI configuration space and finds that it's a bus controller device (SCSI bus controller). Hence, the system BIOS will call its PCI init routine to initialize the SCSI bus.

These procedure probably a dangerous procedure, so it has to be carried-out very carefully. However, our experience shows that it works perfectly in our testbed without causing any damages.

4. Closing

This paper have proved that it's possible to build a low cost x86 embedded system teaching tool by exploiting "flaw" in the PCI specification and PnP BIOS specification. The usability of our teaching tool described here can be improved by developing emulator in Linux for testing the expansion ROM binary developed by the students. We are looking forward to do research on such an emulator in the future.

5. References

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.

1. Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*: Intel Corporation, 2004.
2. Advanced Micro Devices Inc. , *BIOS and Kernel Developer's Guide for the AMD Athlon™ 64 and AMD Opteron™ Processors Rev. 3.08 January 2004*: Advanced Micro Devices Inc. , 2004.
3. Darmawan Mappatutu Salihun, *Award BIOS Reverse Engineering*: The Code-Breakers Journal Vol. 1 No. 2, http://www.CodeBreakers-Journal.com, 2004.
4. Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation, *Plug and Play BIOS Specification Version 1.0A*: Compaq Computer Corporation, Phoenix Technologies Ltd., Intel Corporation, 1994.
5. PCI Special Interest Group, *PCI Local Bus Specification Production Version Revision 2.1*: PCI Special Interest Group, 1995.

Copyright 2005 by the author and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited. This article can be found at http://www.CodeBreakers-Journal.com.