Award BIOS Reverse Engineering

Author: Darmawan Mappatutu Salihun

Abstract

The purpose of this article is to clean up the mess and positioned as a handy reference for myself and the reader as we are going through the BIOS disassembling session. I'm not held responsible about the correctness of any explanation in this article, you have to cross-check what I wrote here and what you have in your hand. Note that what I explain here based on award bios version 4.51PGNM which I have. You can check it against award bios version 6.0PG or 6.0 to see if it's still valid. I'll working on that version when I have enough time. As an addition, I suggest you to read this article througly from beginning to end to get most out of it.

Contents

Award BIOS Reverse Engineering .......................................................... 1
    Author: Darmawan Mappatutu Salihun ............................................. 1
Contents .................................................................................................. 1
1. Foreword ......................................................................................... 2
1. Prerequisite ...................................................................................... 3
1.1. PCI BUS ..................................................................................... 3
1.2. ISA BUS .................................................................................... 6
2. Some Hardware "Peculiarities" .......................................................... 6
3. Some Software "Peculiarities" ............................................................ 9
4. Our Tools of Trade ........................................................................ 13
5. Award BIOS File Structure ............................................................ 14
6. Disassembling the BIOS ................................................................. 18
   6.1. Bootblock ............................................................................... 18
   6.2. System BIOS a.k.a Original.tmp .............................................. 37
1. Foreword

I would like to welcome you to the darkside of a working example of spaghetti code, The Award BIOS. This article is not an official guide to award bios reverse engineering nor it's compiled by an Award Corp. insider. I'm just an ordinary curious person who really attracted to know how my computer BIOS works. I made this article available to the public to share my findings and looking for feedback from others since I'm sure I've made some "obscure mistakes" that I didn't realize during my reverse engineering process. There are several possibilities that make you reading this article now, perhaps you are an "old-time BIOS hacker", perhaps you are a kind of person who really love "system programming" like me or you are just a curious person who like to tinker. One thing for sure, you'll get most of out of this article if you've done some BIOS hacking before and looking forward to improve your skill. However, I've made a prerequisite section below to ensure you've armed yourself with knowledge needed to get most out of this article.

You may be asking, why would anyone need this guide ? indeed, you need this guide if you found yourself cannot figure out how award BIOS code works. In my experience, unless you are disassembling a working BIOS binary, you won't be able to comprehend it. Also, you have to have the majority (if not all) of your mainboard chips datasheets. The most important one is the chipset datasheet.

The purpose of this article is to clean up the mess and positioned as a handy reference for myself and the reader as we are going through the BIOS disassembling session. I'm not held responsible about the correctness of any explanation in this article, you have to cross-check what I wrote here and what you have in your hand. Note that what I explain here based on award bios version 4.51PGNM which I have. You can check it against award bios version 6.0PG or 6.0 to see if it's still valid. I'll working on that version when I have enough time. As an addition, I suggest you to read this article throughly from beginning to end to get most out of it.
1. Prerequisite

First, I would like to thank to the readers of the earlier "beta-version" of this article, from whom I consider that this part of the article should be included.

I have to admit that BIOS is somehow a state of the art code that requires lots of low level x86 knowledge that only matter to such a small audience such as operating system developer, BIOS developer, driver writer, possibly exploit and virus writer (yes exploit and virus writer! coz they are curious people). Due to this fact, there are couple of things that I won't explain here and it's your homework that you should do to comprehend this guide. They are:

- The most important thing is you have to be able to program and understand x86 assembly language. If you don't know it, then you'd better start learning it. I'm using masm syntax throughout this article.
- Protected mode programming. You have to learn how to switch the x86 machine from real mode to protected mode. This means you need to learn a preliminary x86 protected mode OS development. I've done it in the past, that's why I know it pretty good. You can go to [www.osdever.net](http://www.osdever.net) and other x86 operating system developer site to get some tutorials to make yourself comfortable. The most important thing to master is how the protected mode data structures work. I mean how Global Descriptor Table (GDT), Interrupt Descriptor Table (IDT), also x86 control and segment registers work. BIOS, particularly award BIOS uses them to perform its "magic" as later explained in this article.
- What x86 "Unreal-Mode" is. Some people also call these mode of operation "Voodoo-mode" or "Flat real-mode ". It's an x86 state that's between real-mode and protected-mode. This is partially explained below.
- x86 "direct hardware programming". You need to know how to program the hardware directly, especially the chips in your motherboard. You can practice this from within windows by developing an application that directly access the hardware. This is not a must, but it's better if you master it first. You also have to know some x86 bus protocol, such as PCI and ISA. I'll explain a bit about the bus protocols below.
- You have to be able to comprehend part (if not all) of the datasheets of your motherboard chip. Such as the of the northbridge and southbridge control registers.

1.1. PCI BUS

We'll begin with the PCI bus. I've been working with this stuff for quite a while. The official standard for the PCI bus system is maintained by a board named PCISIG (PCI Special Interest Group). This board actually is some sort of cooperation between Intel and some other big corporation such as Microsoft. Anyway, in the near future PCI bus will be fully replaced by a much more faster bus system such as Arapahoe (PCI-Express a.k.a PCI-e) and Hypertransport. But PCI will still remain a standard for sometime I think. I've read some of the specification of the Hypertansport bus, it's backward compatible with PCI. This means that the addressing scheme will remains the same or at least only needs a
minor modification. This also holds true for the Arapahoe. One thing I hate about this PCI stuff is that the standard is not an open standard thus, you gotta pay a lot to get the datasheets and whitepapers. This become my main reason providing you with this sort of tute.

First, PCI bus is a bus which is 32 bits wide. This imply that communicating using this bus should be in 32 bits mode, pretty logical isn't it? So, writing or reading to this bus will require 32 bits 'variable'.

Second, this bus system is defined in the port **CF8h - CFBh** which acts as the address port and port **CFCh - CFFh** which acts as the data port. The role of both ports will be clear soon.

Third, this bus system force us to communicate with them with the following algorithm:

1. Send the address of the part of the device you're willing to read/write at first. Only after that you're access to send/receive data through the data port to/from the device will be granted.
2. Send/receive the data to be read/write through the data port.

As a note, as far as I know every bus/communication protocol implemented in chip design uses this algorithm to communicate with other chip.

With the above definition, now I'll provide you with an x86 assembly code snippet that shows how to use those ports.

<table>
<thead>
<tr>
<th>No.</th>
<th>Mnemonic (masm syntax)</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pushad</td>
<td>save all the contents of General Purpose Registers</td>
</tr>
<tr>
<td>2</td>
<td>mov eax,80000064h</td>
<td>put the address of the PCI chip register to be accessed in eax (offset 64h device 00:00:00 or hostbridge)</td>
</tr>
<tr>
<td>3</td>
<td>mov dx,0CF8h</td>
<td>put the address port in dx. Since this is PCI, we use <strong>CF8h</strong> as the port to open an access to the device.</td>
</tr>
<tr>
<td>4</td>
<td>out dx,eax</td>
<td>send the PCI address port to the I/O space of the processor</td>
</tr>
<tr>
<td>5</td>
<td>mov dx,0CFCh</td>
<td>put the data port in dx. Since this is PCI, we use <strong>CFCh</strong> as the data port to communicate with the device.</td>
</tr>
<tr>
<td>6</td>
<td>in eax,dx</td>
<td>put the data read from the device in eax</td>
</tr>
<tr>
<td>7</td>
<td>or eax, 00020202</td>
<td>modify the data (this is only example, don't try this in your machine, it may hang or even destroy your machine)</td>
</tr>
<tr>
<td>8</td>
<td>out dx,eax</td>
<td>send it back ....</td>
</tr>
<tr>
<td>9</td>
<td>....................</td>
<td>-</td>
</tr>
<tr>
<td>10</td>
<td>popad</td>
<td>pop all the saved register</td>
</tr>
<tr>
<td>11</td>
<td>ret</td>
<td>return...</td>
</tr>
</tbody>
</table>
I think the code above clear enough. In line one the current data in the processors general purpose registers were saved. Then comes the crucial part. As I said above, PCI is 32 bits bus system hence we have to use 32 bits chunk of data to communicate with them. We do this by sending the PCI chip a 32 bits address through eax register, and using port CF8 as the port to send this data. Here's an example of the PCI register (sometimes called offset) address format. In the routine above you saw:

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

the **80000064h** is the address. The meaning of these bits are:

<table>
<thead>
<tr>
<th>bit position</th>
<th>binary value</th>
<th>hexadecimal value</th>
</tr>
</thead>
<tbody>
<tr>
<td>31-0</td>
<td>1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0</td>
<td>8 0 0 0 0 0 0 6 4</td>
</tr>
</tbody>
</table>

- The 31st bit is an enable bit. If this bit sets, it means that we are granted to do a write/read transaction through the PCI bus, otherwise we're prohibited to do so, that's why we need an 8 in the leftmost hexdigit.
- Bits 30 - 24 are reserved bits.
- Bits 23 - 16 is the **PCI Bus number**.
- Bits 15 - 11 is the **PCI Device number**.
- Bits 10 - 8 is the **PCI Function Number**.
- Bits 7 - 0 is the **offset address**.

Now, we'll examine the previous value, that was sent. If you're curious, you'll find out that **80000064h** means we're communicating with the device in bus 0, device 0, and at offset 64. Actually this is the memory controller configuration register of my mainboard's Northbridge. In most circumstances the PCI device that occupy bus 0, device 0, function 0 is the Hostbridge, but you'll need to consult your chipset datasheet to verify this. This stuff is pretty easy to be understood, isn't it? The next routines are pretty easy to understand. But if you still feel confused you'd better learn assembly language a bit, since I'm not here to teach you assembly :( ( . But, in general they do the following jobs: reading the offset data then modifying it then writing it back to the device, if not better to say tweaking it :).
1.2. ISA BUS

AFAIK, ISA bus is not a well standardized bus. Thus, any ISA device can reside virtually almost anywhere in the system's 16-bit I/O address space. My experience with ISA bus is very limited. I've only play with two chips this time around, the first is the CMOS chip and the second one is my mainboard's hardware monitoring chip, i.e. Winbond W83781D. Both chips uses the same "general algorithm" as mentioned above in the PCI BUS, i.e.:

1. Send the address of the part of the device you're willing to read/write at first. Only after that you're access to send/receive data through the data port to/from the device will be granted.
2. Send/receive the data to be read/write through the data port.

My hardware monitoring chip defines port \textbf{295h} as its address port (a.k.a index port) and port \textbf{296h} as its data port. CMOS chip defines port \textbf{70h} as its address port and port \textbf{71h} as its data port.

2. Some Hardware "Peculiarities"

Due to its history, the x86 platform contains lots of hacks, especially its BIOS. This is due to the \textbf{backward compatibility} jargon that should be maintained by any x86 system. In this section I'll try to explain couple of stuff that I found during my BIOS disassembly journey that reveal these peculiarities.

The most important chips which responsible for the BIOS code handling are the southbridge and northbridge. In this respect, the northbridge is responsible for the BIOS shadowing, handling accesses to RAM and BIOS ROM, while the southbridge is responsible for enabling the ROM decode control, which will forward (or not) the memory addresses to be accessed to the BIOS ROM chip. The "special" addresses shown below can reside either in the system DRAM or in BIOS ROM chip, depending on the southbridge and northbridge register setting at the time the BIOS code is executed.

<table>
<thead>
<tr>
<th>Physical Address</th>
<th>Used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>000E 0000h - 000F FFFFh</td>
<td>1 Mbit, 2 MBit, and 4 MBit BIOSes</td>
</tr>
<tr>
<td>000C 0000h - 000D FFFFh</td>
<td>2 MBit, and 4 MBit BIOSes</td>
</tr>
<tr>
<td>0008 0000h - 000B FFFFh</td>
<td>4 MBit BIOSes</td>
</tr>
</tbody>
</table>

The address shown above contain the BIOS code and pretty much system specific, so you have to consult your datasheets to understand it. Below is an example of the VIA693A chipset system memory map.
Table 4. System Memory Map

<table>
<thead>
<tr>
<th>Space</th>
<th>Start</th>
<th>Size</th>
<th>Address Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOS</td>
<td>0</td>
<td>640K</td>
<td>00000000-0009FFFF</td>
<td>Cacheable</td>
</tr>
<tr>
<td>VGA</td>
<td>640K</td>
<td>128K</td>
<td>000A0000-000BFFFF</td>
<td>Used for SMM</td>
</tr>
<tr>
<td>BIOS</td>
<td>768K</td>
<td>16K</td>
<td>000C0000-000C3FFF</td>
<td>Shadow Ctrl 1</td>
</tr>
<tr>
<td>BIOS</td>
<td>784K</td>
<td>16K</td>
<td>000C4000-000C7FFF</td>
<td>Shadow Ctrl 1</td>
</tr>
<tr>
<td>BIOS</td>
<td>800K</td>
<td>16K</td>
<td>000C8000-000CBFFF</td>
<td>Shadow Ctrl 1</td>
</tr>
<tr>
<td>BIOS</td>
<td>816K</td>
<td>16K</td>
<td>000CC0000-000CFFFF</td>
<td>Shadow Ctrl 1</td>
</tr>
<tr>
<td>BIOS</td>
<td>832K</td>
<td>16K</td>
<td>000D0000-000D3FFF</td>
<td>Shadow Ctrl 2</td>
</tr>
<tr>
<td>BIOS</td>
<td>848K</td>
<td>16K</td>
<td>000D4000-000D7FFF</td>
<td>Shadow Ctrl 2</td>
</tr>
<tr>
<td>BIOS</td>
<td>864K</td>
<td>16K</td>
<td>000D8000-000DBFFF</td>
<td>Shadow Ctrl 2</td>
</tr>
<tr>
<td>BIOS</td>
<td>880K</td>
<td>16K</td>
<td>000DC0000-000DFFFF</td>
<td>Shadow Ctrl 2</td>
</tr>
<tr>
<td>BIOS</td>
<td>896K</td>
<td>64K</td>
<td>000E0000-000EFFFF</td>
<td>Shadow Ctrl 3</td>
</tr>
<tr>
<td>BIOS</td>
<td>960K</td>
<td>64K</td>
<td>000F0000-000FFFFF</td>
<td>Shadow Ctrl 3</td>
</tr>
<tr>
<td>Sys</td>
<td>1MB</td>
<td>—</td>
<td>00100000-DRAM Top</td>
<td>Can have hole</td>
</tr>
<tr>
<td>Bus</td>
<td>D Top</td>
<td>—</td>
<td>DRAM Top-FFFEFFFF</td>
<td></td>
</tr>
<tr>
<td>Init</td>
<td>4G-64K</td>
<td>64K</td>
<td>FFFEFFFF-FFFFFFFF</td>
<td>000Fxxxx alias</td>
</tr>
</tbody>
</table>

The most important thing to take into account here is the address aliasing, as you can see the FFFEFFFFh- FFEFFFFFh address range is an alias into 000Fxxxxh, this is where the BIOS ROM chip address mapped (at least in my mainboard, cross check with yours). But, we also have to consider that this only applies at the very beginning of boot stage (just after reset). After the chipset reprogrammed by the BIOS, this address range will be mapped into system DRAM chips. We can consider this as the Power-On default values.

Some "obscure" hardware port which sometimes not documented in the chipset datasheets. Note that this info I found from Intel ICH5 and VIA 586B datasheet.

data sheet.

I/O Port address                  Purpose
92h                              Fast A20 and Init Register
4Doh                             Master PIC Edge/Level Triggered (R/W)
4D1h                             Slave PIC Edge/Level Triggered (R/W)

Table 146. RTC I/O Registers (LPC I/F-D31:F0)

<table>
<thead>
<tr>
<th>I/O Port Locations</th>
<th>If U128E bit = 0</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAM) Index Register</td>
<td>Also alias to 72h and 76h</td>
<td>Real-Time Clock (Standard</td>
</tr>
<tr>
<td>RAM) Target Register</td>
<td>Also alias to 73h and 77h</td>
<td>Real-Time Clock (Standard</td>
</tr>
<tr>
<td>Register (if enabled)</td>
<td></td>
<td>Extended RAM Index</td>
</tr>
<tr>
<td>Register (if enabled)</td>
<td></td>
<td>Extended RAM Target</td>
</tr>
</tbody>
</table>

NOTES:
1. I/O locations 70h and 71h are the standard ISA location for the real-time clock. The map for this bank is shown in Table 147. Locations 72h and 73h are for accessing the extended RAM. The extended RAM bank is also accessed using an indexed scheme. I/O address 72h is used as the address pointer and I/O address 73h is used as the data register. Index addresses above 127h are not valid. If the extended RAM is not needed, it may be disabled.
2. Software must preserve the value of bit 7 at I/O addresses 70h. When writing to this address, software must first read the value, and then write the same value for bit 7 during the sequential address write. Note that port 70h is not directly readable. The only way to read this register is through Alt Access mode. If the NMI# enable is not changed during normal operation, software can alternatively read this bit once and then retain the value for all subsequent writes to port 70h.

The RTC contains two sets of indexed registers that are accessed using the two separate Index and Target registers (70/71h or 72/73h), as shown in Table 147.

<table>
<thead>
<tr>
<th>Index</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>Seconds</td>
</tr>
<tr>
<td>01h</td>
<td>Seconds Alarm</td>
</tr>
<tr>
<td>02h</td>
<td>Minutes</td>
</tr>
<tr>
<td>03h</td>
<td>Minutes Alarm</td>
</tr>
<tr>
<td>04h</td>
<td>Hours</td>
</tr>
<tr>
<td>05h</td>
<td>Hours Alarm</td>
</tr>
<tr>
<td>06h</td>
<td>Day of Week</td>
</tr>
<tr>
<td>07h</td>
<td>Day of Month</td>
</tr>
<tr>
<td>08h</td>
<td>Month</td>
</tr>
<tr>
<td>09h</td>
<td>Year</td>
</tr>
<tr>
<td>0Ah</td>
<td>Register A</td>
</tr>
<tr>
<td>0Bh</td>
<td>Register B</td>
</tr>
<tr>
<td>0Ch</td>
<td>Register C</td>
</tr>
<tr>
<td>0Dh</td>
<td>Register D</td>
</tr>
<tr>
<td>0Eh–7Fh</td>
<td>114 Bytes of User RAM</td>
</tr>
</tbody>
</table>

There are couples of more things to take into account, such as the Video BIOS and other expansion ROM handling. I'll try to cover this stuff next time when I have done dissecting BIOS code that handle it.
3. Some Software "Peculiarities"

There are couples of tricky areas in the BIOS code due to the execution of some of its parts in ROM. I'll present some of my findings below.

**call** instruction is not available during bios code execution from within BIOS ROM chip. This is due to **call** instruction uses/manipulate stack while we don't have **writeable** area in BIOS ROM chip to be used for stack. What I mean by manipulating stack here is the "implicit" push instruction which is executed by the **call** instruction to "write/save" the return address in the stack. As we know clearly, address pointed to by **ss:sp** at this point is in ROM, meaning: **we can't write into it.** If you think, why don't use the RAM altogether? the DRAM chip is not even available at this point. It hasn't been tested by the BIOS code, thus **we haven't know if RAM even exists!**

The peculiarity of **retn** instruction. There is macro that's called **ROM_call** as follows:

```assembly
ROM_CALL MACRO RTN_NAME
    LOCAL RTN_ADD
    mov sp,offset DGROUP:RTN_ADD
    jmp RTN_NAME
RTN_ADD:  dw DGROUP:$+2
ENDM
```

an example of this macro "in action" as follows:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:6000</td>
<td>66 B8 00 00 00 80</td>
<td>mov eax, 80000000h  ; copy offset addr to ax</td>
</tr>
<tr>
<td>F000:6008</td>
<td>24 FC</td>
<td>and al, 0FCh</td>
</tr>
<tr>
<td>F000:600A</td>
<td>BA F8 0C</td>
<td>mov dx, 0CF8h</td>
</tr>
<tr>
<td>F000:600D</td>
<td>66 EF</td>
<td>out dx, eax</td>
</tr>
<tr>
<td>F000:600F</td>
<td>B2 FC</td>
<td>mov dl, 0FCh</td>
</tr>
<tr>
<td>F000:6011</td>
<td>0A D1</td>
<td>or dl, cl</td>
</tr>
<tr>
<td>F000:6013</td>
<td>EC</td>
<td>in al, dx</td>
</tr>
<tr>
<td>F000:6014</td>
<td>C3</td>
<td>retn</td>
</tr>
</tbody>
</table>

from Procedure

F000:6014 F000_6000_read_pci_byte_endp

......
F000:6043 18 00 GDTR_F000_6043 dw 18h  ; limit of GDTR (3 valid desc entry)
F000:6045 49 60 0F 00 dd 0F6049h  ; GDT physical addr (below)
F000:6049 00 00 00 00 00 00 00 00 dq 0  ; null descriptor
F000:6051 FF FF 00 00 0F 9F 00 00 dq 9F0F0000FFFFh  ; code descriptor:
F000:6051 = F 0000h; limit=FFFFh; DPL=0;

F000:6051
exec/ReadOnly, conforming, accessed;
F000:6051
granularity=byte; Present; 16-bit segment
F000:6059 FF FF 00 00 93 8F 00 dq 8F93000000FFFFh ; data
descriptor:
F000:6059 ; base addr
= 00h; seg_limit=F FFFFh; DPL=0;
F000:6059 ; Present; read-write, accessed;
F000:6059 ;
granularity = 4 KByte; 16-bit segment
......
F000:619B 0F 01 16 43 60        lgdt qword ptr GDTR_F000_6043 ; Load
Global Descriptor Table Register
F000:61A0 0F 20 C0              mov   eax, cr0
F000:61A3 0C 01                 or    al, 1           ; set FMode
flag
F000:61A5 0F 22 C0              mov   cr0, eax
F000:61A8 EA AD 61 08 00         jmp   far ptr 8:61ADh ; jmp below in
16-bit PMode (abs addr F 61ADh)
F000:61A8 ; (code
segment with base addr = F 0000h)
F000:61AD                       ; -----------------------------------
F000:61AD B8 10 00              mov   ax, 10h         ; load ds with
valid data descriptor
F000:61B0 8E D8                 mov   ds, ax          ; ds = data
descriptor (GDT 3rd entry)
......
F000:61BC B9 6B 00              mov   cx, 6Bh         ; DRAM
arbitration control
F000:61BF BC C5 61              mov   sp, 61C5h
F000:61C2 E9 3B FE              jmp   F000_6000_read_pci_byte ; Jump
F000:61C2 ; -----------------------------------
F000:61C5 C7 61                 dw 61C7h
F000:61C7 ; -----------------------------------
F000:61C7 0C 02                 or    al, 2           ; enable VC-
DRAM

as you can see, you have to take into account that the retn instruction is
affected by the current value of ss:sp register pair, but ss register is not even
loaded with "correct" 16-bit protected mode value prior to using it! how this code
even works ? the answer is a bit complicated. Let's look at the last time ss
register value was manipulated before the code above executed:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:E060</td>
<td>8C C8</td>
<td>mov ax, cs</td>
</tr>
</tbody>
</table>
| F000:E062 | 8E D0      | mov ss, ax ; ss = cs (ss =
 F000h a.k.a F_segment) |
| F000:E064 |            | assume ss:F000 |

Note: this routine is executed in real-mode
as you can see, ss register is loaded with F000h (the current BIOS code 16-bit segment in real-mode). This code implies that the hidden descriptor cache register (that exist for every selector/segment register) is loaded with "ss * 16" or F0000h physical address value. And this value is retained even when the machine is switched into 16-bit Protected Mode above since ss register is not reloaded. A snippet from Intel Software Developer Manual Vol.3:

8.1.4. First Instruction Executed
The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFF0H. This address is 16 bytes below the processor’s uppermost physical address. The EPROM containing the software-initialization code must be located at this address.
The address FFFFFFF0H is beyond the 1-MByte addressable range of the processor while in real-address mode. The processor is initialized to this starting address as follows. The CS register has two parts: the visible segment selector part and the hidden base address part. In real address mode, the base address is normally formed by shifting the 16-bit segment selector value 4 bits to the left to produce a 20-bit base address. However, during a hardware reset, the segment selector in the CS register is loaded with F000H and the base address is loaded with FFFFFFF0H. The starting address is thus formed by adding the base address to the value in the EIP register (that is, FFFF0000 + FFF0H = FFFFF00H).
The first time the CS register is loaded with a new value after a hardware reset, the processor will follow the normal rule for address translation in real-address mode (that is, [CS base address = CS segment selector * 16]). To insure that the base address in the CS register remains unchanged until the EPROM based software-initialization code is completed, the code must not contain a far jump or far call or allow an interrupt to occur (which would cause the CS selector value to be changed).

also a snippet from DDJ (Doctor Dobbs Journal):

At power-up, the descriptor cache registers are loaded with fixed, default values, the CPU is in real mode, and all segments are marked as read/write data segments, including the code segment (CS).
According to Intel, each time the CPU loads a segment register in real mode, the base address is 16 times the segment value, while the access rights and size limit attributes are given fixed, "real-mode compatible" values. This is not true. In fact, only the CS descriptor cache access rights get loaded with fixed values each time the segment register is loaded - and even then only when a far jump is encountered. Loading any other segment register in real mode does not change the access rights or the segment size limit attributes stored in the descriptor cache registers. For these
segments, the access rights and segment size limit attributes are honored from any previous setting (see Figure 3). Thus it is possible to have a four giga-byte, read-only data segment in real mode on the 80386, but Intel will not acknowledge, or support this mode of operation.

If you want to know more about descriptor cache and how it works, you can search the web for articles about "descriptor cache" or "x86 unreal mode", the most comprehensive guide can be found in one of Doctor Dobbs Journal and Intel Software Developer Manual Vol.3 chapter 3 Protected Mode Memory Management in section 3.4.2 Segment Registers. Back to our ss register, now you know that the "actor" here is the descriptor cache register, especially its base address part. The visible part of ss is only a "place holder" and the "register in-charge" for the "real" address calculation/translation is the hidden descriptor cache. Whatever you do to this descriptor cache will be in effect when any code, stack or data value addresses are translated/calculated. In our case, we have to use "stack segment" with "base address" at F0000h physical address in 16-bit protected mode. This is not a problem, since the base address part of ss descriptor cache register already filled with F0000h in one of the code above. This explains why the code above can be executed flawlessly. Another example:

```
Address     Hex          Mnemonic
F000:61BF   BC C5 61    mov   sp, 61C5h
F000:61C2   E9 3B FE    jmp   F000_6000_read_pci_byte ; Jump
F000:61C2

-------------------------------
F000:61C5   C7 61       dw   61C7h
```

in this code we have to make ss:sp points to F61C5h for retn instruction to work. Indeed, we've done it, since ss contains F0000h (its descriptor cache base address part) and as you can see, sp contains 61C5h, the physical address pointed to by ss:sp is F0000h + 61C5h which is F61C5h physical address.
4. Our Tools of Trade

You are only as good as your tools. Yeah, this also holds true here. To begin the journey, we'll need a couple of tools as follows:

1. IDA Pro disassembler. I'm using IDA Pro version 4.50. You can use your favourite interactive disassembler. I found IDA Pro is the most suitable for me. We need an interactive disassembler since the BIOS binary that we're going to disassemble is not a trivial code. At some points of its execution it resides in ROM, hence, no stack available. It uses some sort of stack trick to do procedure/routine calling.

2. A good hex editor. I'm using HexWorkshop ver. 3.0b. The most beneficial feature of this hex editor is it's capability to calculate checksums for the selected range of file that we open inside of it.

3. LHA 2.55, it's needed if you want to modify the bios binary. Or, you can use winzip or another compression/decompression program that can handle LZH/LHA file if you only want to get the compressed bios components.

4. Some bios modification tools i.e. : CBROM, I'm using version 2.08, 2.07 and 1.24 and MODBIN. There are two types of modbin, modbin6 for award bios ver. 6 and modbin 4.50.xx for award bios ver. 4.5xPGNM. We need these tools to look at the bios components much more easily. You can download it at www.biosmods.com, in the download section.

5. Some chipset datasheets. This depends on the mainboard bios binary that you're gonna dissect. Some datasheets available at www.rom.by. I'm dissecting a VIA693A-596B mainboard. I have the datasheets at my hand, except for the southbridge i.e. VIA596B, which is substituted by VIA586B and 686A datasheet, since the complete VIA596B datasheet is not available.

6. Intel Software Developer Manual Volume 1, 2 and 3. These are needed since BIOS sometimes uses "exotic" instruction set. Also, there are some system data structures that are hard to remember and need to be looked up, such as GDT and IDT.

OK, now we're armed. What we need to do next is to understand the basic stuff by using the hex editor before proceeding through the disassembling session.
5. Award BIOS File Structure

Award BIOS file consists of several components. Some of the components are LH level-1 compressed. We can recognize them by looking at the "-lh5-" signature in the beginning of that component using hex editor. Here's an example:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>25F2</td>
<td>%-lh5-........W...</td>
</tr>
<tr>
<td>00000010</td>
<td>0000</td>
<td>..A ..awardext.r</td>
</tr>
<tr>
<td>00000020</td>
<td>6F6D</td>
<td>om.t ......#I.</td>
</tr>
</tbody>
</table>

Besides the compressed components, there are also some "pure" 16-bit x86 binary components. Award BIOS execution begins at this "pure" binary (uncompressed) components.

We have to know the entry point to start our disassembly to this BIOS binary. We know that the execution of x86 processor begins in 16-bit real mode at address F000:FF0 (physical address FFFF FFF0) following restart or power up, as per Intel Software Developer Manual Vol.3 "System Programming". Based on our intuition, this address must contain a 16-bit real mode x86 executable code. That's true. Below is the "memory map" of award bios binary that I have. It's a 2MBit/256 KB bios image for Iwill VD133 mainboard.

- **The compressed components**:
  1. **0000h - 3AACh**: XGROUP ROM (awardext.rom), this is an award extension rom. It contains routine that is called from the system BIOS, i.e. original.tmp
  2. **3AADh - 97AFh**: CPUCODE.BIN, this is the microcode for the BIOS.
  3. **97B0h - A5CFh**: ACPITBL.BIN, the acpi table.
  4. **A5D0h - A952h**: Iwill.bmp, the BMP logo.
  5. **A953h - B3B1h**: nнопrom.bin, I haven't know yet what this component's role.
  6. **B3B2h - C86Ch**: Antivir.bin, the bios bootsector antivirus.
  7. **C86Dh - 1BEDCh**: ROSUPD.BIN, this is a custom bios component in my bios. It's used to display a customized Boot Logo and indicator
  8. **20000h - 35531h**: original.tmp, this is the system BIOS. This component located in this address in most award bioses, but sometimes also located in the very beginning of the bios binary, i.e. 0000h.
Note:

- Between the compressed ROSUPD.BIN and original.tmp there are padding FFh bytes. These padding bytes also found after the compressed original.tmp and the pure binary BIOS components that will be explained below. An example of these padding bytes:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00037D00</td>
<td>2A42 4253 532A 0060 0070 0060 0060 00A0</td>
<td><em>BBSS</em>.p..p..p..</td>
</tr>
<tr>
<td>00037D10</td>
<td>3377 4670 8977 ACCF C4CF 0100 00FF FFFF</td>
<td>3wFp.w.........</td>
</tr>
<tr>
<td>00037D20</td>
<td>FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF</td>
<td>...............</td>
</tr>
<tr>
<td>00037D30</td>
<td>FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF</td>
<td>...............</td>
</tr>
</tbody>
</table>

- The compressed component can be extracted easily by copying and pasting it into a new binary file in Hexworkshop. Then, decompress this new file by using LHA 2.55 or winzip. If we are into using winzip, give the new file an ".lzh" extension so that it'll be automatically associated with winzip. Recognizing where we should "cut" to get the new file is pretty easy, just look for the "-lh5-" string. Two bytes preceding "-lh5-" string is the beginning of the file and the end of the file is always 00h, right before the next compressed file (with the "-lh5-" marker in its beginning), right before the padding bytes or right before some kind of checksum. I present two examples below, the highlighted bytes is the beginning or the end of the compressed file.

**compressed CPUCODE.BIN file in my BIOS:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>00003AA0</td>
<td>4E61 19E6 9775 2B46 BA55 85F0 0024 382D</td>
<td>Na...u+F.U...88-</td>
</tr>
<tr>
<td>00003AB0</td>
<td>6C68 352D DC5C 0000 00A0 0000 0000 0140 lh5-...@</td>
<td></td>
</tr>
<tr>
<td>00003AC0</td>
<td>2001 0B43 5055 434F 4445 2E42 494E BCAA</td>
<td>..CPUCODE.BIN..</td>
</tr>
<tr>
<td>00003ADD</td>
<td>2000 0038 3994 9700 52C4 A2CF F040 0000</td>
<td>..88...R......@</td>
</tr>
<tr>
<td>00003AE0</td>
<td>4000 0000 0000 0000 0000 0000 0000 0000</td>
<td>@.............</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000097A0</td>
<td>0E3C 8FA7 FFF4 FFFE 9FFF D3FF FFFB FF00</td>
<td>.&amp;lt.............</td>
</tr>
<tr>
<td>000097B0</td>
<td>24D9 2D6C 6835 2DFA 0D00 00A6 2100 0000</td>
<td>$.-lh5-......!</td>
</tr>
</tbody>
</table>

**compressed ORIGINAL.TMP file in my BIOS:**

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>ASCII</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001FFFE</td>
<td>FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF</td>
<td>...............</td>
</tr>
<tr>
<td>00020000</td>
<td>251A 2D6C 6835 2D09 5501 0000 0002 0000</td>
<td>$.-lh5-....U....</td>
</tr>
</tbody>
</table>
00020010 0000 5020 010C 6F72 6967 696E 616C 2E74 ..P
..original.t
00020020 6D70 0CD9 2000 002D 7888 F0FD D624 A5BA mp.. ..-
x....$..
........
00035510 019E 6E67 BF11 8582 88D9 4E7C BEC8 C34C ..ng......N|...
00035520 401D 189F BDD0 A176 17F0 4383 1D73 BF99 @......v..C..s..
00035530 00C9 FFFF FFFF FFFF FFFF FFFF FFFF FFFF ............
00035540 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF .............

• The pure binary components:
  0. 36000h - 36C4Ah : Memory sizing routine, this routine also initialize the Host Bridge and the CPU/RAM clock in my BIOS
  1. 37000 - 37D1Ch : The decompression block, this routine contains the LZH decompression engine which decompresses the compressed bios components above.
  2. 3C000h - 3CFE4h : This area contains various routine, the lower 128KB BIOS address decode enabler, the default VGA initialization (executed if system bios is erratic), Hostbridge initialization routine, etc.
  3. 3E000h - 3FFFFh : This area contains the Boot Block code.

Note: in between some of the components lies padding bytes. Some are FFh bytes and some are 00h bytes.

• The memory map in the real system (mainboard).
We have to note that the memory map above is described as we see the BIOS binary in a hex editor. In the mainboard BIOS chip, it's a bit different and more complex. It's mapped in my mainboard as follows (it's maybe a bit different with yours, consult your chipset documentation):
  0. 0000h - 3FFFFh in the BIOS binary (as displayed in hex editor) is mapped into FFFC 0000h - FFFF FFFFh in my system memory space. Due to my system's northbridge (as per its datasheet), address FFFF 0000h - FFFF FFFFh is just an alias to F 0000h - FF FFFFh or speaking in "real-mode lingo" F000:0000h - F000:FFFFh. Note that this mapping only applies just after power-on, since it's the chipset's power-on default value. It's not guaranteed to be valid after the chipset is reprogrammed by the BIOS itself. There are some other "kludge" though and they are really system dependent. You have to consult Intel Software Developer Manual Volume 3 (system programming) and your chipset datasheet.
  1. Due to the explanation in 1. , the pure binary BIOS components is mapped as follows (note: just after power-on):
     • BootBlock : F000:E000h - F000:FFFFh
     • Decompression Block : F000:7000h - F000:7D1Ch
     • Early Memory Initialization : F000:6000h - F000:6C4Ah
2. The compressed BIOS components are mapped into system memory space after they are decompressed in a different manner. They reliant on the decompression block routine, but there are few mappings that seem remain the same across different BIOS files. These mappings are (as per my BIOS. Yours may differ, but the segment address very possibly the same):

- original.tmp a.k.a System BIOS : $E000:0000h - $F000:5531h
- awardext.rom a.k.a Award extension ROM : $4100:0000h - $4100:xxxxh. Later relocated to $6000:0000h - $6000:xxxxh by original.tmp, before it's executed.

We have to be aware of this mapping during our journey.

---

**Note:**
It's very easy to get lost due to the sheer complexity of the BIOS binary address mapping into the real system. But, there are some guidelines that will ease our effort during our disassembly session using IDA Pro as follows:

- Begin the disassembly session with the pure binary components. I just copy my BIOS file at $36000h - $3FFFFh to get these components and paste it into a new binary file to be disassembled. We need these components to reside in one file since they are inter-related each other. Then I disassemble this new file by setting its address mapping in IDA Pro to $F000:6000h - $F000:FFFFh and disabling segment naming so that I can see its "real-mode address" in the system during its execution.
- Decompress the system bios (original.tmp) somewhere, you'll find that its size is 128KB. Then disassemble it by setting its address mapping in IDA Pro to $E000:0000h - $F000:FFFFh. The address mapping should be like that since this compressed bios component is decompressed by the decompression block somewhere in memory and then relocated into this address range before it's "jumped-into" by the bootblock code (gets executed). AFAIK, this mapping apply to all award BIOS. Also remember to disable segment naming, so that we can see its "real-mode address" in the system during its execution.
6. Disassembling the BIOS

Due to Intel System Programming Guide I mentioned before, we'll begin the disassembly session at address F000:FFF0h (note: look at the memory mapping above and adjust IDA Pro to suit it). You may ask: *How the hell this is even possible?* Intel Software Developer Manual Vol. 3 (PROCESSOR MANAGEMENT AND INITIALIZATION - First Instruction Executed) says:

The first instruction that is fetched and executed following a hardware reset is located at physical address FFFFFFFOH. The answer is: I repeat that my northbridge chipset aliases address range **FFFF FFFFh - FFFF FFFFh** to **000Fxxxxh**. Also, note that the southbridge has no means to alter the translation of this address range. It just passes the addresses directly to the BIOS ROM chip. Hence, there's no difference between address **FFFF FFF0h** and **F FFF0h** (or **F000:FF00** in "real-mode lingo") just after power-on or reset. It's that simple heh ;) . This is the BootBlock area, it always contains a far jump into the bootblock routine, mostly to **F000:E05Bh**. From this point we can continue the disassembly to cover the majority of the pure binary part. In reality, lots of the pure binary code is never executed at all since it's very seldom your system BIOS gets corrupted and the Bootblock POST (Power On Self Test) routine takes place.

6.1. Bootblock

From this point we can disassemble the bootblock routines. Now, I'll present some of the "obscure" areas of the BIOS code in the disassembled bootblock. This is with respect to my BIOS, yours may vary but it will be very similar.

At Virtual Shutdown routine:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:E07F BC 0B F8</td>
<td>BC 0B F8</td>
<td>mov sp, 0F80Bh</td>
<td>contains E103h (memory presence test code)</td>
</tr>
<tr>
<td>E103h</td>
<td></td>
<td>jmp Ct_Very_Early_Init; return</td>
<td>from this jump</td>
</tr>
<tr>
<td>F000:E082</td>
<td>E9 7B 15</td>
<td>jmp Ct_Very_Early_Init; return</td>
<td>is redirected to F000:E103h</td>
</tr>
</tbody>
</table>

At Reset PCI Bus routine:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:E1A0 BF A6 E1</td>
<td>BF A6 E1</td>
<td>mov di, 0E1A6h</td>
<td>the return</td>
</tr>
<tr>
<td>addr of the jump below</td>
<td></td>
<td>jmp Reset_PCI_Bus</td>
<td></td>
</tr>
<tr>
<td>F000:E1A3 E9 42 99</td>
<td>E9 42 99</td>
<td>jmp Reset_PCI_Bus</td>
<td></td>
</tr>
<tr>
<td>Jumpless_in Decompress_Area,</td>
<td></td>
<td>Program CPU</td>
<td></td>
</tr>
<tr>
<td>F000:E1A3</td>
<td>clock pin, host clock</td>
<td>Program CPU</td>
<td></td>
</tr>
<tr>
<td>F000:E1A3</td>
<td>jumperless platform ???</td>
<td>for</td>
<td></td>
</tr>
<tr>
<td></td>
<td>....</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
F000:7CDD                   _delay:                  ; CODE XREF:
Reset_PCI_Bus+1F5            ; Loop while
F000:7CDD  E2  FE           loop _delay
CX != 0
F000:7CDF FF E7             jmp    di
F000:E1A3h

At call to memory detection routine:

Address   Hex                                Mnemonic
F000:E1D6                          Checksum is ok , execute memory
detection
F000:E1D6  2E  8B  07                mov    ax, cs:[bx] ; ax = cs:[bx]
(Fs:[7D06h] is 6000h)
F000:E1D9  25  00  F0                and    ax, 0F000h ; ax = 6000h
F000:E1DC  8B  F0                    mov    si, ax ; si = 6000h
F000:E1DE  81  C6  FC  0F            add    si, 0FFCh ; add
si,MEMORY_PRESENCE_OFFSET; si=6FFCh
F000:E1E2  2E  8B  34                mov    si, cs:[si] ; si = 60B4h
F000:E1E5  BC  EC  E1                mov    sp, 0E1ECh ; pointer to
pointer to ret addr below
F000:E1E8  FF  E6                    jmp    si ; jmp to
F000:60B4h, execute memory detection
F000:E1E8                           returns at
F000:E1F8

This code gets executed before the bootblock is copied to RAM. In case the RAM is faulty, the system will halt and output error code from system speaker.

At bootblock get copied and executed in RAM:

Address   Hex                                Mnemonic
F000:E2AA                          ;----------- Enter 16-bit Protected
Mode (Flat) ----------------------
F000:E2AA                           assume ds:F000
F000:E2AA  0F 16 F6 E4             lgdt    qword ptr GDTR_F000_E4F6 ;
Load Global Descriptor Table

Register
F000:E2AF  0F 20 C0                mov    eax, cr0
F000:E2B2  0C 01                    or    al, 1 ; activate
FMode flag
F000:E2B4  0F 22 C0                mov    cr0, eax
F000:E2B7  EB  00                    jmp    short $+2 ; clear
prefetch, enter 16-bit FMode. We're
F000:E2B7                          ; using the
"unchanged" hidden value of CS
F000:E2B7                          ; register
F000:E2B7                          (descriptor cache) from previous
F000:E2B7                          ; "FMode
session" in memory_check_routine
F000:E2B7                          ; for code
segment desc
F000:E2B9  B9  08  00                mov    ax, 8
F000:E2BC  8E D8                    mov    ds, ax ; ds = 1st
entry in GDT loaded above
F000:E2BE                           assume ds:nothing
F000:E2BE  8E C0                    mov    es, ax ; es = 1st
entry in GDT loaded above
F000:E2C0
There are two locations to access
chipsets can not access onboard ROM
use the space on ISA bus. To
change address to 0FFFE0000H
to read BIOS contents at 0E0000H

assume es:nothing
mov esi, 0E0000h  ; starting
addr of compressed original.tmp
cmp dword ptr [esi+2], '5hl-' ;
LHA signature
(ZF=1)
or esi, 0FF00000h ; esi =
FFFE0000h
--- move entire BIOS (i.e. original.tmp
and bootblock) from ROM at E0000h-FFFFFh to RAM at
10000h-2FFFFh --
LHA_sign_OK:
CODE XREF:
mov edi, 10000h  ; buffer at
1000:0
mov ecx, 8000h  ; copy 128
KByte to buffer (original.tmp &
bootblock)
rep movs dword ptr es:[edi], dword
ptr [esi] ; Move Bytes from
String to String
xor ax, cr0
and al, 0FEh  ; clear PMode
bit
mov cr0, eax
jmp short $+2  ; clr
prefetch, back to RealMode
jmp far ptr 2000h:0E2F7h ; Jump
below in RAM
-------------
Setup temporary stack at 0:1000H, at
this point
Bios code (last 128 Kbyte) is still
compressed
except the bootblock and
decompression code
BootBlock_in_RAM:  ; ax = 0000h
xor ax, ax
mov ss, ax  ; ss = 0000h
assume ss:nothing
mov sp, 1000h  ; ss:sp =
The last 128KB of BIOS code (E000:0000h - F000:FFFFh) get copied to RAM as follows:

1. Northbridge power-on default values aliases F0000h-FFFFFh address space with FFFE FFFFh-FFFF FFFFh, where the BIOS ROM chip address space mapped. That's why the following code is safely executed:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:FFF0</td>
<td>EA 5B E0 00 F0</td>
<td>jmp far ptr entry_point;</td>
</tr>
<tr>
<td>Northbridge is responsible for decoding address of this jump into BIOS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F000:FFF0</td>
<td>; chip through address aliasing. So, even if</td>
<td></td>
</tr>
<tr>
<td>F000:FFF0</td>
<td>; this is a far jump (read Intel Software Guide Vol.3 for info)</td>
<td></td>
</tr>
<tr>
<td>F000:FFF0</td>
<td>; we are still in BIOS chip d00d ;)</td>
<td></td>
</tr>
<tr>
<td>F000:FFF0</td>
<td>; vi693A: FFFEFFFFFF-FFFFFFFF is 000Fxxxx alias.</td>
<td></td>
</tr>
</tbody>
</table>

   also, northbridge power-on default values disables DRAM shadowing for this address space. Thus, read/write to this address space will not be forwarded to DRAM. At the same time, there's no control register in southbridge that controls the mapping of this address space. Hence, I suspect that read operation to this address space will be "directly forwarded" to the BIOS ROM chip without being altered by the southbridge. Of course this read operation first pass through northbridge which apply the address aliasing scheme.

2. Very close to the beginning of Bootblock execution, routine Ct_Very_Early_Init executed. This routine reprogram the PCI-to-ISA bridge (in southbridge) to enable decoding of address E0000h-FFFFFh to ROM, i.e. forwarding read operation in this address space into the BIOS ROM chip. The northbridge power-on default values disables DRAM shadowing for this address space. Thus, read/write to this address space will not be forwarded to DRAM.

3. Then comes the routine displayed above which copied the last 128KB BIOS ROM chip content (address E0000h - FFFFh) into DRAM at 1000:0000h - 2000:FFFFh and continues execution from there. This can be accomplished since this address space is mapped only to DRAM by the chipset, no special address translation.

4. From this point on, Bootblock code execution is within segment 2000h in RAM. This fact holds true for all Bootblock routines explained below. Note that the segment address shown in bootblock routines below uses segment F000h. It should be segment 2000h but I hadn't change it. Pay attention to this!
At call to bios decompression routine and the jump into decompressed system bios:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:E3DC</td>
<td>E8 33 01</td>
<td>call Expand_BIOS</td>
</tr>
<tr>
<td>F000:E3DF</td>
<td>EB 03</td>
<td>jmp short BIOS_chksum_OK</td>
</tr>
<tr>
<td>F000:E3E1</td>
<td></td>
<td>; checksum is good</td>
</tr>
<tr>
<td>F000:E3E1</td>
<td></td>
<td>; ---------------------------------</td>
</tr>
<tr>
<td>F000:E3E1</td>
<td></td>
<td>BIOS_chksum_err: ; CODE XREF:</td>
</tr>
<tr>
<td>F000:E3E4</td>
<td></td>
<td>BIOS_chksum_OK: ; CODE XREF:</td>
</tr>
<tr>
<td>F000:E3DF</td>
<td></td>
<td>if checksum ok</td>
</tr>
<tr>
<td>F000:E3E4</td>
<td>8E D8</td>
<td>; setup source</td>
</tr>
<tr>
<td>F000:E3E4</td>
<td></td>
<td>for shadowing</td>
</tr>
<tr>
<td>F000:E3E4</td>
<td></td>
<td>; so, if ok,</td>
</tr>
<tr>
<td>F000:E3E6</td>
<td></td>
<td>ds = 5000h</td>
</tr>
<tr>
<td>F000:E3E6</td>
<td>B0 C5</td>
<td>assume ds: nothing</td>
</tr>
<tr>
<td>F000:E3E8</td>
<td>E6 80</td>
<td>out 80h, al</td>
</tr>
<tr>
<td>F000:E3E8</td>
<td>B0 00</td>
<td>mov ax, 1000h</td>
</tr>
<tr>
<td>F000:E3E9</td>
<td>80 00</td>
<td>mov al, 0C5h</td>
</tr>
<tr>
<td>F000:E3EC</td>
<td></td>
<td>purchasable bit</td>
</tr>
</tbody>
</table>

The source data segment is 5000H if checksum is good.
And segment 1000H is for shadowing original BIOS image if checksum is bad. BIOS will shadow bootblock and boot from it.

Address F80DH is shadowed by decompressed code (i.e. original.bin and others),
And "BootBlock_POST" will be executed if checksum is bad.

F000:E3F2  EA 0D F8 00 F0           jmp far ptr F000_segment ; jump to F000 segment
during execution of Expand_BIOS routine, the compressed BIOS code (original.tmp) at 1000:0000h - 2000:FFFFh in RAM decompressed into E000:0000h - F000:FFFFh also in RAM. Note that the problem due to address aliasing and DRAM shadowing are handled during the decompression by setting the appropriate chipset registers. Below is the basic run-down of what this routine accomplished:

2. Enable FFF80000h-FFFFFFFFFFh decoding. Access to this address will be forwarded into the BIOS ROM chip by the PCI-to-ISA Bridge. PCI-to-ISA bridge ROM decode control register is in-charge here. This is needed, since my BIOS is 256KB and only 128KB of it has been copied into RAM, i.e. the original.tmp and bootblock which is at 1000:0000h-2000:FFFFh by now.

3. Copy lower 128KB of BIOS code from FFFC0000h-FFFFFFFFFFh in ROM chip into 8000:0000h - 9000:FFFFh in DRAM.

4. Disable FFF80000h-FFFFFFFFFFh decoding. Access to this address will not be forwarded into the BIOS ROM chip by the PCI-to-ISA Bridge.

5. Verify checksum of the whole compressed BIOS image, i.e. calculate the 8-bit checksum of copied compressed BIOS image in RAM (i.e. 8000:0000h - 9000:FFFFh + 1000:0000h - 2000:FFFFh) and compare the result against result stored in 2000:7FFDh. If 8-bit checksum doesn't match, then goto BIOS_chksum_err, else continue to decompression routine.

6. Look for the decompression engine by looking for *BBSS* string in segment 2000h, then execute the decompression routine for all of the compressed BIOS components.

7. Decompress the compressed BIOS components. Note that at this stage only original.tmp and it's extension i.e. awardext.rom (probably also awardyt.rom, I haven't verify it) which get decompressed. The other component treated in different fashion. The BootBlock_expand routine only process their decompressed/expansion area information then put it somewhere in RAM. We need some preliminary info before delving into this step as follows:

- The format of the LZH level-1 compressed bios components. The address ranges where these BIOS components will be located after decompression are contained within this format. The format is as follows (it applies to all compressed components):

<table>
<thead>
<tr>
<th>Offset from 1st byte</th>
<th>Offset in Real Header</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>00h</td>
<td>N/A</td>
<td>The header length of the component. It depends on the file/component name.</td>
</tr>
<tr>
<td>01h</td>
<td>N/A</td>
<td>The header 8-bit checksum, not including the first 2 bytes (header length and header checksum byte).</td>
</tr>
<tr>
<td>Address Range</td>
<td>Value Ranges</td>
<td>Description</td>
</tr>
<tr>
<td>---------------</td>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>02h - 06h</td>
<td>00h - 04h</td>
<td>LZH Method ID (ASCII string signature). In my BIOS it's &quot;-lh5-&quot; which means: 8k sliding dictionary (max 256 bytes) + static Huffman + improved encoding of position and trees.</td>
</tr>
<tr>
<td>07h - 0Ah</td>
<td>05h - 08h</td>
<td>Compressed file/component size in little endian dword value, i.e. MSB at 0Ah and so forth</td>
</tr>
<tr>
<td>0Bh - 0Eh</td>
<td>09h - 0Ch</td>
<td>Uncompressed file/component size in little endian dword value, i.e. MSB at 0Eh and so forth</td>
</tr>
<tr>
<td>0Fh - 10h</td>
<td>0Dh - 0Eh</td>
<td>Decompression offset address in little endian word value, i.e. MSB at 10h and so forth. The component will be decompressed into this offset address (real mode addressing is in effect here).</td>
</tr>
<tr>
<td>11h - 12h</td>
<td>0Fh - 10h</td>
<td>Decompression segment address in little endian word value, i.e. MSB at 12h and so forth. The component will be decompressed into this segment address (real mode addressing is in effect here).</td>
</tr>
<tr>
<td>13h</td>
<td>11h</td>
<td>File attribute. My BIOS components contain 20h here, which is normally found in LZH level-1 compressed file.</td>
</tr>
<tr>
<td>14h</td>
<td>12h</td>
<td>Level. My BIOS components contain 01h here, which means it's a LZH level-1 compressed file.</td>
</tr>
<tr>
<td>15h</td>
<td>13h</td>
<td>Component filename name length in byte.</td>
</tr>
<tr>
<td>16h - [15h+filename_len]</td>
<td>14h - [13h+filename_len]</td>
<td>Component filename (ASCII string)</td>
</tr>
<tr>
<td>[16h+filename_len] -</td>
<td>[14h+filename_len] -</td>
<td>File/component CRC-16 in little endian word value, i.e. MSB at [HeaderSize - 2h] and so forth.</td>
</tr>
<tr>
<td>[17h+filename_len]</td>
<td>[15h+filename_len]</td>
<td>Operating System ID. In my BIOS it's always 20h (ASCII</td>
</tr>
</tbody>
</table>
space character) which don't resemble any LZH OS ID known to me.

<table>
<thead>
<tr>
<th>19h+filename_len</th>
<th>17h+filename_len</th>
<th>[1Ah+filename_len]</th>
<th>[18h+filename_len]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Next header size.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>In my BIOS it's</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>always 0000h which</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>means no extension</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>header.</td>
<td></td>
</tr>
</tbody>
</table>

- **Note:**
  - The left-most offset is calculated from the beginning of the compressed component and the contents description "addressing" with respect to the 1st byte of the component. The "offset in Real Header" is used within the "scratch-pad RAM" explained below.
  - Each component is terminated with EOF byte, i.e. 00h byte.
  - In my BIOS, there are ReadHeader procedure which contains routine to read and verify the content of this header. One of the key "procedure call" there is a call into *FreadCRC*, which reads the bios component header into a "scratch-pad" RAM area beginning at 3000:0000h (ds:0000h). This scratch-pad area is filled with the "real-LZH-header value" which doesn't include the first 2 bytes (header size and header 8-bit checksum), but includes the 3rd byte (offset 02h) until offset HeaderSize+02h.
  - The location of various checksums which are checked prior and during the decompression process.

<table>
<thead>
<tr>
<th>Location</th>
<th>Calculation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Right after compressed original.tmp</td>
<td>original.tmp 8-bit checksum. This value is calculated after it's copied to RAM at segment 1000h and 2000h. The code as follows:</td>
</tr>
</tbody>
</table>

- **Address** Assembly Code
  - F000:E307 ;BIOS checksum verify
  - F000:E307 ;Now, the 128Kb BIOS (0E0000H-0FFFFFH) is in 10000H-2FFFFH.
  - F000:E307 mov ax, 1000h ; point to 0E000H bios segment
  - F000:E30A mov ds, ax ; ds = 1000h (E000h segment of the BIOS
  - F000:E30A ; copied to RAM)
  - F000:E30C assume ds:nothing
  - F000:E30C mov bx, ds:9 ; size over 64Kb ; equ---> bx = 0001h
  - F000:E310 mov cx, ds:7 ; get compressed size; equ---> cx = 5509h
  - F000:E314 add cl, ds:0 ; add header size; equ---> 25h + 09h = 2Eh
  - F000:E318 adc ch, 0 ; Add with Carry
  - F000:E31B adc bx, 0 ; Add with Carry |
F000:E31E add cx, 3 ; add
cx,TAIL_BYTE_SIZE;
F000:E31E          ;
COMPRESSED_SIZE = 552Eh + 3h = 5531h
F000:E31E          ; This
is the remainder of the cmprrssd
F000:E31E          ;
original.tmp in seg_F000h
F000:E321 adc bx, 0  ; Add
with Carry
F000:E324          ;
below_or_equ_64Kb ; jmp
if compressed size less than 64Kb
F000:E326          ;
size remainder in next 64KB
F000:E326          ;
(8seg_F000h)
F000:E326          ;
F000:E328          ;
xor cx, cx          ; code
size to sum_up for 1st 64Kb
F000:E328          ;
(cx=0000h means 64KB)
F000:E32A          ;
F000:E32A below_or_equ_64Kb: ; CODE
XREF: F000:E324
F000:E32A          ;
xor si, si          ; si =
0000h
F000:E32C          ;
xor ah, ah          ; ah =
00h (initial 8-bit chksum)
F000:E32E          ;
F000:E32E add_next_byte: ; CODE
XREF: F000:E331 F000:E343
F000:E32E          ;
load next 64KB string
F000:E331          ;
loop add_next_byte ; loop
while cx != 0 (<64KB)
F000:E333          ;
F000:E333          ;
F000:E333          ;
F000:E335          ;
F000:E335          ;
F000:E335          ;
F000:E335          ;
F000:E337          ;
F000:E337          ;
F000:E337          ;
F000:E339          ;
F000:E339          ;
F000:E33A          ;
F000:E33A          ;
This is the 8-bit checksum of the decompression engine which starts at F000:7000h (2000:7000h after copied to RAM) in my BIOS. The code as follows:

### Address | Assembly Code
--- | ---
F000:E35E | Verify checksum of decompress engine
F000:E35E | mov ds, ax ; ds = 2700h (2000:7000h)
F000:E360 | assume ds:nothing ; ds = F000h segment in RAM
F000:E360 | xor ah, ah ; ah = 0000h
F000:E362 | xor si, si ; si = 0000h
F000:E364 | mov cx, 0FFFh ; 4095 Byte boundary
F000:E364 | chksum_loop: ; CODE
F000:E364 | lodsb ; Load String
F000:E366 | add ah, al ; calc 8 bit checksum
F000:E36A | loop chksum_loop ; Loop while CX != 0
F000:E36C | cmp ah, [si] ; decompression engine checksum OK?
F000:E36E | jnz BIOS_cxsm_error ; jump if no

Right after the decompression engine

This is the 8-bit checksum of all compressed BIOS plus the 8-bit checksum of the decompression engine (not including its previously calculated checksum above). The code:

### Address | Assembly Code
--- | ---
F000:E512 | call Extern_execute1 ; copy lower 128 KByte bios code from ROM
F000:E512 | ; (at FFFC 0000h - FFFD 0000h) to RAM
F000:E512 | ; (at 8000:0000h-9000:FFFFh)
F000:E515 | xor ah, ah ; ah = 00h
F000:E517 | xor cx, cx ; cx = 0000h
F000:E519 | mov bx, 8000h
F000:E51C  mov   ds, bx      ; ds = 8000h, contains compressed
F000:E51C                                      ; lower 128KB bios components (awdext,etc.)
F000:E51E  assume ds:nothing
F000:E51E  xor   si, si      ; si = 0000h
F000:E520
F000:E520  next_seg8000h_byte: ; CODE
XREF: Expand_Bios+11 Expand_Bios+1F
F000:E520  lodsb             ; Load String
F000:E521  add   ah, al      ; calc 8-bit chksum, result placed at ah
F000:E523  loop  next_seg8000h_byte ; loop while cx != 0, i.e. 64 KByte
F000:E525
F000:E525  mov   bx, ds     ; bx = ds
F000:E527  cmp   bh, 90h    ; 64 KByte chksum-ed ?
F000:E52A  jnb   _8000h_chksum_done ; yes
F000:E52C  add   bh, 10h    ; no, continue calc-ing in next segment
F000:E52C                                      ; we're calc-ing 128KByte code chksum
F000:E52F  mov   ds, bx
F000:E531  assume ds:nothing
F000:E531  jmp   short
next_seg8000h_byte ; Jump
F000:E533 ; ---------------------------------------------
F000:E533  _8000h_chksum_done: ; CODE
XREF: Expand_Bios+18
F000:E533  mov   bx, 1000h  ; 1000h, 1st 64KB BIOS img (E000h seg of compressed original.tmp)
F000:E533                                      ; compressed
F000:E536  mov   ds, bx      ; ds = 1000h
F000:E538  assume ds:nothing
F000:E538  xor   si, si      ; si = 0000h
F000:E53A  cld
Direction Flag
F000:E53B
F000:E53B  next_seg1000h_byte: ; CODE
XREF: Expand_Bios+2C Expand_Bios+3B
F000:E53B  lodsb             ; Load String
F000:E53C  add   ah, al      ; calc 8 bit chksum, contd from chksum above
F000:E53E  loop  next_seg1000h_byte ; Loop while CX != 0
F000:E540
F000:E540  cmp   bh, 20h    ; is 64KB reached? (seg_F000 reached?)
F000:E543  jnb   _1000h_chksum_done ; yes
F000:E545  add   bh, 10h    ; no, proceed calc-ing in next segment
F000:E548  mov   ds, bx

F000:E54A   assume ds:nothing
F000:E54A   mov   cx, 7FFEh ; calc
seg_F000 chksum only until 7FFEh
F000:E54D   jmp   short
next_seg1000h_byte ; Jump
F000:E54F ; ---------------------------------
F000:E54F
F000:E54F_1000h_chksum_done: ; CODE
XREF: Expand_Bios+31
F000:E54F   cmp   ah, [si]    ; cmp calc-
ed chksum and chksum
F000:E54F                     ; pointed
to by [si] (at F000:7FFEh, i.e. B2h)
F000:E54F   jnz   BIOS_cksm_error ; Jump
F000:E54F                     ; if Not Zero (ZF=0)

The following are the key parts of the decompression routine:

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:E512</td>
<td>Expand_Bios proc near</td>
</tr>
</tbody>
</table>

..........
F000:E555 | mov bx, 0  ; mov bx,Temp_VGA_Seg |
F000:E558    | mov es, bx  ; es = 0000h |
F000:E55A | assume es:nothing |
F000:E55A | mov word ptr es:7004h, 0FFFFh ; mov word es:[Temp_VGA_Off+4],ffffh |
F000:E561 | F000:E561 | xor al, al ; clr expand flag |
F000:E563 | mov bx, 1000h |
F000:E566 | move es, bx  ; es = 1000h; |
SrcSegment,i.e. seg_E000h
F000:E566 | ; |
F000:E568 | assume es:nothing |
F000:E568 | xor bx, bx  ; bx = 0000h ; |
SrcOffset
F000:E56A | call BootBlock_Expand ; read compressed original.tmp header and |
F000:E56A | to segment 5000h |
F000:E56A | from its LZH header |
F000:E56A | ; on return |
ecx=total_component_cmprssd_size
F000:E56D | jb decompression_error ; Jump if Below |
(FP=1) |
F000:E56F | test ecx, 0FFFF0000h ; ecx & FFFF 0000h ;check against wrong |
original.tmp size, i.e. < 64 KB |
F000:E576 | jz decompression_error ; Jump if Zero |
(ZF=1) |
F000:E578 | mov bx, 2000h |

F000:E57B   mov   es, bx          ; es = 2000h;SrcSegment, i.e. seg_F000h
F000:E57D   assume es:nothing
F000:E57D   mov   bx, 1           ; chksum byte size
F000:E580   jmp   short Expand_else ; Jump
............
F000:E59D   expand_else:           ; CODE XREF:
Expand_Bios+6E Expand_Bios+99
F000:E59D   add   bx, cx          ; es = 2000h (seg_F000h
in RAM)
F000:E59D   bx =
offset_after_original.tmp+chksum;
F000:E59D   ; this input likely
return CF=1 since
F000:E59D   ; it isn't a LZH
compressed component
F000:E59F   call  BootBlock_Expand ; Call Procedure
F000:E5A2   jb    Expand_else_Over ; Jump if Below (CF=1)
F000:E5A4   test  ecx, 0FFFF0000h ; Logical Compare
F000:E5AD   Expand_else_Over:     ; CODE XREF:
Expand_Bios+89 Expand_Bios+90
F000:E5AD   call  Extern_execute2 ; expand lower 128KB
BIOS code (C0000h-DFFFFh)
F000:E5AD   ; this routine only
decompress awardext.rom, other
F000:E5AD   ; component only get
their ExpSegment processed
F000:E5B0   jz    BIOS_cksm_error ; jump if zero
(awardext.rom not found)
F000:E5B4   mov   ax, 5000h       ; ax = 5000h on success
F000:E5B7   clc                   ; Clear Carry Flag
F000:E5B8   retn                  ; Return Near from
Procedure
F000:E5B8   Expand_Bios endp

F000:E5B9   BootBlock_Expand proc near
F000:E5B9   cmp  dword ptr es:[bx+0Fh], 40000000h ; 1st
addr contain 5000 0000h
F000:E5B9   decomp_Seg:Offset equ
4000 0000h ?
F000:E5B9   ; (is extension
component ?)
F000:E5C2   jnz   not_40000000h
No,skip; at first
this jump is taken
............
F000:E5EA   not_40000000h:        ; CODE XREF:
BootBlock_Expand+9
F000:E5EA   mov   dx, 3000h       ; mov dx,Exp_Data_Seg;
decomp scratch pad?
F000:E5ED   push  ax
F000:E5EE   push  es
F000:E5EF   call  Search_BBSS_label ; on return si =
7D06h
F000:E5EF   (cs:di = 2000:7D06h -
- bios in ram)
F000:E5F2   pop   es
F000:E5F3   assume es:nothing
F000:E5F3   push  es
F000:E5F4   mov   ax, es         ; ax = 1000h (1st pass)

F000:E5F6    shr   ax, 0Ch  ; ax = 1h
F000:E5F9    mov   es, ax  ; es = 1h
F000:E5FB    assume es:nothing
F000:E5FB    mov   ax, cs:[si+0Eh] ; mov ax, 7789h (addr of decompression code)
F000:E5FF    call  ax    ; call 7789h i.e Expand (decompression engine)
F000:E601    pop   es  ; es = 1000h
F000:E602    assume es:nothing
F000:E602    pop   ax
F000:E603    retn     ; Return Near from Procedure

F000:E603 BootBlock_Expand endp

F000:7789 ; Code below is called from Bootblock_Expand procedure
F000:7789 ; (at F000:E5FF) and should return there when finished.
F000:7789 Expand proc near

..........
F000:780E    add   bx, 12h  ; bx = 12h
F000:7811    call  Get_Exp_Src_Byte ; get es:[bx+12h] to AL (ExpSegment hi byte)
F000:7814    sub   bx, 12h  ; restore bx value (first pass = 0000h)
F000:7817    cmp   al, 40h  ; is "extension component" ?
F000:7817    (original.tmp)
F000:7817    (awardext.rom)
F000:7817    components: al equ 40h
F000:7817    ; at all other
F000:7817    caveat is here d00d !!!
F000:7819    jnz   Not_POST_USE ; jmp if no: for original.tmp and awadext.rom
F000:7819    ; goto decompress, otherwise no
F000:781B    add   bx, 11h  ; bx = ExpSegment_lo_byte index
F000:781E    call  Get_Exp_Src_Byte ; al = ExpSegment_lo_byte
F000:7821    sub   bx, 11h  ; restore bx
F000:7824    or    al, al  ; segment 4000h ?
F000:7826    jnz   Record_to_buffer ; jmp if no
F000:7826    ; (all "extension component" jump here)

..........
F000:7830   Record_to_buffer:    ; CODE XREF: Expand+9D
F000:7830   movzx  dx, al  ; dx = ExpSegment_lo_byte
F000:7833   inc   bx  ; bx = header_chksum_index
F000:7834   call  Get_Exp_Src_Byte ; al = header_chksum
F000:7837   sub   al, dl  ; al = header_chksum - ExpSegment_lo_byte
F000:7839   call  Set_Exp_Src_Byte ; header_chksum = al
F000:783C   dec   bx  ; restore bx
F000:783D   xor   al, al  ; al = 00h

Copyright © 2004 and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited.
F000:783F  add   bx, 11h         ; bx = ExpSegmenet_lo_byte
F000:7842  call  Set_Exp_Src_Byte ; ExpSegment_lo_byte = 00h (ExpSegment=4000h)
F000:7845  sub   bx, 11h         ; restore bx
F000:7848  inc   dx              ; dx = ExpSegment_lo_byte + 1
F000:7849  shl   dx, 2           ; dx = 4*(ExpSegment_lo_byte + 1)
F000:784C  add   di, dx          ; di = 6000h + dx (look above!)
F000:784E  mov   gs:[di], bx     ; 0000:[di] = CmpressedCompnnt_offset_addr
F000:7851  mov   cx, es          ; cx = ExpSegment
F000:7853  mov   gs:[di+2], cx   ; 0000:[di+2]=ExpSegment
F000:7857  call  Get_Exp_Src_Byte ; al = header_len
F000:785A  movzx ecx, al         ; ecx = header_len
F000:785E  add   bx, 7           ; bx --> point to compressed file size
F000:7861  call  Get_Exp_Src_Dword ; eax = compressed file size
F000:7864  sub   bx, 7           ; restore bx
F000:7867  add   ecx, eax        ; ecx = header_len + compressed_file_size
F000:786A  add   ecx, 3          ; ecx = total_compressed_component_size
F000:786E  pop   gs              ; restore gs
F000:7870  assume gs:nothing
F000:7873  jmp   exit_proc       ; Jump
F000:787B  jb    exit_proc       ; error, something wrong (CF=1)
F000:787F  mov   ax, ds:108h     ; mov ax,ExpSegment
F000:7882  mov   ds:104h, ax     ; mov TgtSegment,ax
F000:7885  mov   ax, ds:10Ah     ; mov ax,ExpOffset
F000:7888  mov   ds:106h, ax     ; mov TgtOffset,ax
F000:788B  ;--calculate compressed total size and return when decompress complete
F000:788B  mov   ecx, ds:310h    ; mov ecx,compsize
F000:7890  xor   eax, eax       ; eax = 0000 0000h
F000:7893  mov   al, ds:571Ch    ; mov al,headersize; compressed header size
F000:7896  add   ecx, eax       ; Add
F000:7899  add   ecx, 3          ; add ecx,COMPRESSED_UNKNOWN_BYTE;
F000:7899  mov   edx, ds:314h    ; mov edx,origsize

Copyright © 2004 and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited.
F000:78A2  push  edx
F000:78A4  push  ecx  ; save ecx (total compressed component size)
F000:78A6  push  bx  ; bx = 0000h
F000:78A7  add  bx,  5  ; offset 5 ('-lh0-' or '-lh5-')
F000:78AA  call  Get_Exp_Src_Byte ; get compress or store type value
F000:78AD  pop  bx  ; bx = 0000h (1st pass)
F000:78AE  cmp  al, '0'  ; is it "-lh0-"? first pass is no jump taken
                      ..........
F000:78E1  Not_Store: ; CODE XREF: Expand+127
F000:78E1  push  word ptr ds:104h ; push word ptr TgtSegment
F000:78E5  push  word ptr ds:106h ; push word ptr TgtOffset
F000:78E9  push  large dword ptr ds:314h ; push dword ptr origsize
F000:78EE ; extract content from compressed file
F000:78EE  call  Extract ; call LZH decompression routine
F000:78F1  pop  dword ptr ds:314h ; pop dword ptr origsize
F000:78F6  pop  word ptr ds:106h ; pop word ptr TgtOffset
F000:78FA  pop  word ptr ds:104h ; pop word ptr TgtSegment
F000:78FE  Expand_Over: ; CODE XREF: Expand+156
F000:78FE  call  ZeroFill_32K_mem ; zero fill 32K in segment pointed by ds
                      ..........
F000:7901  pop  ecx  ; ecx = "total compressed size" (restore ecx)
F000:7903  pop  edx
F000:7905  clc  ; decompression success
F000:7906  exit_proc: ; CODE XREF: Expand+E7 Expand+F2
F000:7906  pop  es
F000:7907  pop  bx
F000:7908  pop  eax
F000:790A  retn  ; Return Near from Procedure
F000:790A  Expand_endp
8. After looking at these exhaustive list of hints, we managed to construct the mapping of the decompressed BIOS components as described below:

<table>
<thead>
<tr>
<th>Starting address of decompressed BIOS component in RAM</th>
<th>Compressed Size</th>
<th>Decompressed Size</th>
<th>Decompression State (by Bootblock code)</th>
<th>Component description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4100:0000h</td>
<td>3A85h</td>
<td>57C0h</td>
<td>Decompressed to RAM beginning at address in column one.</td>
<td>awardext.rom, this is a &quot;helper module&quot; for original.tmp</td>
</tr>
<tr>
<td>4001:0000h</td>
<td>5CDCh</td>
<td>A000h</td>
<td>Not yet decompressed</td>
<td>cpucode.bin, this is the CPU microcode</td>
</tr>
<tr>
<td>4003:0000h</td>
<td>DFAh</td>
<td>21A6h</td>
<td>Not yet decompressed</td>
<td>acpitbl.bin, this is the ACPI table</td>
</tr>
<tr>
<td>4002:0000h</td>
<td>35Ah</td>
<td>2D3Ch</td>
<td>Not yet decompressed</td>
<td>iwillbmp.bmp, this is the EPA logo</td>
</tr>
<tr>
<td>4027:0000h</td>
<td>A38h</td>
<td>FECh</td>
<td>Not yet decompressed</td>
<td>mnprom.bin, explanation N/A</td>
</tr>
<tr>
<td>4007:0000h</td>
<td>1493h</td>
<td>2280h</td>
<td>Not yet decompressed</td>
<td>antivir.bin, this is BIOS antivirus code</td>
</tr>
<tr>
<td>4028:0000h</td>
<td>F63Ah</td>
<td>14380h</td>
<td>Not yet decompressed</td>
<td>ROSUPD.bin, seems to be custom Logo display procedure</td>
</tr>
<tr>
<td>5000:0000h</td>
<td>15509h</td>
<td>20000h</td>
<td>Decompressed to RAM beginning at address in column one.</td>
<td>original.tmp, the system BIOS</td>
</tr>
</tbody>
</table>

9. **Note:** The decompression addresses marked with green background are treated in different fashion as follows:
   - It's not the real decompression area of the corresponding component as you can see from the explanation above. It's only some sort of "place holder" for the real decompression area that's later handled by original.tmp. The conclusion is: **only original.tmp and awardext.rom get decompressed by ExpandBios routine in Bootblock.** If you want to verify this, try summing up the decompressed code size, it won't fit!
All of these component's decompressed segment address are changed to 4000h by Expand procedure as you can see in the routine at F000:7842h above.

The 40xxh shown in their "Starting Address ... (for decompression)" actually an ID that works as follows: 40 (hi-byte) is an ID that mark it as an "Extension BIOS" to be decompressed later during original.tmp execution. xx is an ID that will be used in original.tmp execution to refer to the component to be decompressed. This will be explained more thoroughly in original.tmp explanation later.

All of these components are decompressed during original.tmp execution. The decompression result is placed starting at address 4000:0000h, but not at the same time. Some of it (maybe all, I'm not sure yet) also relocated from that address to retain their contents after another component also decompressed in there. More explanation on this available at original.tmp section below.

10. Shadow the BIOS code. Assuming that the decompression routine successfully completed, the routine above then copy the decompressed system BIOS (original.tmp) from 5000:0000h - 6000:FFFFh in RAM to E000h - FFFFh also in RAM. This is accomplished as follows:

1. Reprogram the northbridge shadow RAM control register to enable write only into E000h - FFFFh, i.e. forward write operation into this address range to DRAM (not to the BIOS ROM chip anymore).

2. Perform a string copy operation to copy the decompressed system BIOS (original.tmp) from 5000:0000h - 6000:FFFFh to E000h - FFFFh.

3. Reprogram the northbridge shadow RAM control register to enable read only into E000h - FFFFh, i.e. forward read operation into this address range to DRAM (not to the BIOS ROM chip anymore). This is also to write-protect the system BIOS code.

11. Enable the microprocessor cache then jump into the decompressed system BIOS. This step is the last step in the normal Bootblock code execution path. After enabling the processor cache, the code then jump into the write-protected system BIOS (original.tmp) at F000:F80Dh in RAM as seen in the code above. This jump destination address seems to be the same across different award bioses.

Now, I'll present the "memory map" of the compressed and decompressed BIOS components just before jump into decompressed original.tmp is made. This is important since it will ease us in dissecting the decompressed original.tmp later. We have to note that by now, all code execution happens in RAM, no more code execution from within BIOS ROM chip.
<table>
<thead>
<tr>
<th>Address Range in RAM</th>
<th>Decompression State (by Bootblock code)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:6000h - 0000:6xxxh</td>
<td>N/A</td>
<td>This area contains the header of the extension component (component other than original.tmp and awardext.rom) fetched from the compressed BIOS at 8000:0000h - 9000:FFFFh (previously BIOS component at FFFC0000h - FFFDFFFFh in the BIOS chip). Note that this is fetched here by part of the bootblock in segment 2000h.</td>
</tr>
<tr>
<td>1000:0000h - 2000:5531h</td>
<td>Compressed</td>
<td>This area contains the compressed original.tmp. It's part of the copy of the last 128KB of the BIOS (previously BIOS component at E000:0000h - F000:FFFFh in the BIOS chip). This code is shadowed here by the bootblock in BIOS ROM chip.</td>
</tr>
<tr>
<td>2000:5532h - 2000:5FFFh</td>
<td>N/A</td>
<td>This area contains only padding bytes.</td>
</tr>
<tr>
<td>2000:6000h - 2000:FFFFh</td>
<td>Pure binary (executable)</td>
<td>This area contains the bootblock code. It's part of the copy of the last 128KB of the BIOS (previously BIOS component at E000:0000h - F000:FFFFh in the BIOS ROM chip). This code is shadowed here by the bootblock in BIOS ROM chip. This is where our code currently executing (the &quot;copy&quot; of bootblock in segment 2000h).</td>
</tr>
<tr>
<td>4100:0000h - 4100:57C0h</td>
<td>Decompressed</td>
<td>This area contains the decompressed awardext.rom. Note that the decompression process is accomplished by part of the bootblock in segment 2000h.</td>
</tr>
<tr>
<td>5000:0000h - 6000:FFFFh</td>
<td>Decompressed</td>
<td>This area contains the decompressed original.tmp. Note that the decompression process is accomplished by part of the bootblock in segment 2000h.</td>
</tr>
<tr>
<td>8000:0000h - 9000:FFFFh</td>
<td>Compressed</td>
<td>This area contains the copy of the first/lower 128KB of the BIOS (previously BIOS component at FFFC0000h - FFFD0000h in the BIOS chip). This code is shadowed here by the bootblock in segment 2000h.</td>
</tr>
<tr>
<td>E000:0000h - F000:FFFFh</td>
<td>Decompressed</td>
<td>This area contains copy of the decompressed original.tmp, which is shadowed here by the bootblock in segment 2000h.</td>
</tr>
</tbody>
</table>
The last thing to note is: what I explain about bootblock here only covers the normal Bootblock code execution path, which means I didn't explain about the Bootblock POST that takes place in case original.tmp corrupted. I'll try to cover it later when I have time to dissect it. This is all about the bootblock right now, from this point on we'll dissect the original.tmp.

6.2. System BIOS a.k.a Original.tmp

We'll just proceed as in bootblock above, I'll just highlight the places where the "code execution path" are obscure. So, by now, you're looking at the disassembly of the decompressed original.tmp of my bios.

The entry point from Bootblock:

<table>
<thead>
<tr>
<th>Address</th>
<th>Hex</th>
<th>Mnemonic</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:F80D</td>
<td></td>
<td>This code is jumped into by the</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bootblock code</td>
</tr>
<tr>
<td>F000:F80D</td>
<td>E9 02 F6</td>
<td>if everything went OK</td>
</tr>
<tr>
<td></td>
<td></td>
<td>jmp sysbios_entry_point ;</td>
</tr>
</tbody>
</table>

This is where the bootblock jumps after relocating and write-protecting the system BIOS.

The awardext.rom and extension BIOS components (lower 128KB bios-code) relocation routine:

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>F000:EE12</td>
<td>sysbios_entry_point: ; CODE XREF: F000:F80D</td>
</tr>
<tr>
<td>F000:EE12</td>
<td>mov ax, 0</td>
</tr>
<tr>
<td>F000:EE15</td>
<td>mov ss, ax</td>
</tr>
<tr>
<td>F000:EE17</td>
<td>mov sp, 1000h ; setup stack at 0:1000h</td>
</tr>
<tr>
<td>F000:EE1A</td>
<td>call setup_stack ; Call Procedure</td>
</tr>
<tr>
<td>F000:EE1D</td>
<td>call init_DRAM_shadowRW ; Call Procedure</td>
</tr>
<tr>
<td>F000:EE20</td>
<td>mov si, 5000h ; ds=5000h (look at copy_mem_word)</td>
</tr>
<tr>
<td>F000:EE23</td>
<td>mov di, 0E000h ; es=E000h (look at copy_mem_word)</td>
</tr>
<tr>
<td>F000:EE26</td>
<td>mov cx, 8000h ; copy 64KByte</td>
</tr>
<tr>
<td>F000:EE29</td>
<td>call copy_mem_word ; copy E000h segment routine, i.e.</td>
</tr>
<tr>
<td>F000:EE29</td>
<td>; copy 64Kbyte from 5000:0h to 6000:0h</td>
</tr>
<tr>
<td>E000:0h</td>
<td></td>
</tr>
<tr>
<td>F000:EE2C</td>
<td>call j_init_DRAM_shadowR ; Call Procedure</td>
</tr>
<tr>
<td>F000:EE2F</td>
<td>mov si, 4100h ; at this point 4100h</td>
</tr>
<tr>
<td>F000:EE2F</td>
<td>; ds = XGroup segment decompressed, i.e.</td>
</tr>
<tr>
<td>F000:EE32</td>
<td>mov di, 6000h ; es = new XGroup segment</td>
</tr>
<tr>
<td>F000:EE35</td>
<td>mov cx, 8000h ; copy 64KByte</td>
</tr>
<tr>
<td>F000:EE38</td>
<td>call copy_mem_word ; copy XGroup segment, i.e.</td>
</tr>
<tr>
<td>F000:EE38</td>
<td>; 64Kbyte from 4100:0h to 6000:0h</td>
</tr>
<tr>
<td>F000:EE3B</td>
<td>call Enter_UnrealMode ; jump below in UnrealMode</td>
</tr>
<tr>
<td>F000:EE3E</td>
<td>Begin_in_UnrealMode</td>
</tr>
<tr>
<td>F000:EE3E</td>
<td>mov ax, ds</td>
</tr>
</tbody>
</table>

Copyright © 2004 and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited.
F000:EE40  mov es, ax ; es = ds (3rd entry in GDT)
F000:EE40  ; base_addr=0000 0000h; limit 4GB
F000:EE41  assume es:nothing
F000:EE41  mov esi, 80000h ; mov esi,(POST_Cmprssed_Temp_Seg shl 4)
F000:EE41  ; relocate lower 128KB bios Seg
F000:EE42  mov edi, 160000h
F000:EE44  mov ecx, 8000h
F000:EE45  cld                ; Clear Direction Flag
F000:EE45  rep movs dword ptr es:[edi], dword ptr [esi] ; move
F000:EE45  ; 128k data to 160000h (phy addr)
F000:EE49  call Leave_UnrealMode ; Call Procedure
F000:EE4A  mov byte ptr [bp+214h], 0 ; mov byte ptr
F000:EE4A  ; POST_SPEED[bp],Normal_Boot
F000:EE4B  mov si, 626Bh    ; offset 626Bh (E000h POST tests)
F000:EE4E  push 0E000h        ; segment E000h
F000:EE4F  push si            ; next instruction offset (626Bh)
F000:EE50  retf               ; jmp to E000:626Bh

F000:7440 Enter_UnrealMode proc near ; CODE XREF: F000:EE3B
F000:7440  mov ax, cs
F000:7442  mov ds, ax         ; ds = cs
F000:7444  assume ds:F000
F000:7444  lgdt qword ptr GDTR_F000_5504 ; Load Global Descriptor
F000:7449  mov eax, cr0
F000:744C  or al, 1           ; Logical Inclusive OR
F000:744E  mov cr0, eax
F000:7451  mov ax, 10h
F000:7454  mov ds, ax         ; ds = 10h (3rd entry in GDT)
F000:7456  assume ds:nothing
F000:7456  mov ss, ax         ; ss = 10h (3rd entry in GDT)
F000:7458  assume ss:nothing
F000:7458  retn                ; Return Near from Procedure
F000:7458 Enter_UnrealMode endp

F000:5504 GDTR_F000_5504 dw 30h ; DATA XREF: Enter_PMode+4
F000:5504  ; GDT limit (6 valid desc)
F000:5506  dd 0F550Ah     ; GDT phy addr (below)
F000:550A  dq 0          ; null desc
F000:5512  dq 9F0F0000FFFFh ; code desc (08h)
F000:5512  ;
F000:5512  base_addr=F0000h; seg_limit=64KB; code,execute/ReadOnly
F000:5512  ;
F000:5512  conforming,accessed; granularity=1Byte; 16-bit segment;
F000:5512  ; segment present, code, DPL=0
F000:5512  dq 8F93000000FFFFh ; data desc (10h)
F000:5512  ; base_addr=0000
F000:5512  0000h; seg_limit=4GB; data,R/W,accessed;
F000:5512  ; granularity=4KB; 16-bit segment;
F000:5512  ; segment present,
F000:5512  ; data, DPL=0
F000:5512  dq 0FF0093FF0000FFFFh ; data desc 18h
F000:5512  ;
F000:5512  base_addr=FFFF0000h; seg_limit=64KB; data,R/W,accessed;
F000:5512  ; 16-bit segment, granularity = 1 byte;
F000:5512  ; segment present, data, DPL=0.
Note: after the execution of code above, the "memory map" is changed once again. But this time only for the compressed "BIOS extension" i.e. the lower 128KB of BIOS code and the decompressed awardext.rom, the "memory map" mentioned in the Bootblock explanation above partially overwritten.

<table>
<thead>
<tr>
<th>New Address Range in RAM</th>
<th>Decompression State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>6000:0000h - 6000:57C0h</td>
<td>Decompressed</td>
<td>This is the relocated awardext.rom</td>
</tr>
<tr>
<td>160000h - 17FFFFh</td>
<td>Compressed</td>
<td>This is the relocated compressed &quot;BIOS extension&quot;, including the compressed awardext.rom. (i.e. this is the copy of FFFC0000h - FFFDFFFF in the BIOS rom chip.</td>
</tr>
</tbody>
</table>

At call to the POST routine a.k.a "POST jump table execution".

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>E000:626B</td>
<td>The last of the these POST routines starts the EISA/ISA</td>
</tr>
<tr>
<td>E000:626B</td>
<td>section of POST and thus this call should never return.</td>
</tr>
<tr>
<td>E000:626B</td>
<td>If it does, we issue a POST code and halt.</td>
</tr>
<tr>
<td>E000:626B</td>
<td>This routine called from F000:EE68h</td>
</tr>
<tr>
<td>E000:626B</td>
<td>sysbios_entry_point_contd a.k.a NORMAL_POST_TESTS</td>
</tr>
<tr>
<td>E000:626B</td>
<td>mov cx, 3</td>
</tr>
<tr>
<td>E000:626E</td>
<td>mov di, 61C2h</td>
</tr>
<tr>
<td>E000:6271</td>
<td>call RAM_POST_tests ; this won't return in normal condition</td>
</tr>
<tr>
<td>E000:6274</td>
<td>jmp short Halt_System ; Jump</td>
</tr>
</tbody>
</table>
E000:6276 ; ---------------- SUBROUTINE ------------------------
E000:6276

E000:6276 RAM_POST_tests proc near ; CODE XREF: last_E000_POST+D
E000:6276                         ; last_E000_POST+18 ...
E000:6276 mov al, cl         ; cl = 3
E000:6278 out 80h, al        ; manufacture's diagnostic
E000:6279              ; checkpoint
E000:627A push 0F000h
E000:627D pop fs             ; fs = F000h
E000:627F
E000:627F ; This is the beginning of the call into E000 segment
E000:627F ; POST function table
E000:627F assume fs:F000
E000:627F                         ; in the beginning :
E000:627F mov ax, cs:[di]    ; di = 61C2h ; ax = cs:[di] = 154Eh
E000:627F                         ; called from E000:2489 w/ di=61FCh
E000:627F (dummy)
E000:6282 inc di             ; Increment by 1
E000:6283 inc di             ; di = di + 2
E000:6284 or ax, ax          ; Logical Inclusive OR
E000:6286 jz RAM_post_return ; RAM Post Error
E000:6288 push di             ; save di
E000:6289 push cx             ; save cx
E000:628A call ax            ; call 154Eh (relative call addr)
E000:628A                         ; ,one of this call
E000:628A                         ; won't return in normal condition
E000:628C pop cx             ; restore all
E000:628D pop di             ; di = di + 1
E000:628E jbf RAM_post_return ; Jump if Below (CF=1)
E000:6290 inc cx             ; Increment by 1
E000:6291 jmp short RAM_POST_tests ; Jump
E000:6293 ; ------------------

E000:6293 RAM_post_return:        ; CODE XREF: RAM_POST_tests+10
E000:6293                         ; RAM_POST_tests+18
E000:6293 retn                     ; Return Near from Procedure
E000:6293 RAM_POST_tests endp

E000:61C2 E0_POST_TESTS_TABLE:
E000:61C2 dw 154Eh           ; Restore boot flag
E000:61C4 dw 156Fh           ; Chk Mem Refrsh Toggle
E000:61C6 dw 1571h           ; keyboard (and its controller)
POST
E000:61C8 dw 16D2h           ; chksum ROM, check EEPROM
E000:61C8 dw 1745h           ; on error generate spkr tone
E000:61CA dw 178Ah           ; Check CMOS circuitry
E000:61CC dw 1798h           ; "chipset defaults" initialization
E000:61CE dw 184Bh           ; init CPU cache (both Cyrix and
Intel)
E000:61D0 dw 18B8h           ; init interrupt vector, also
initialize
E000:61D0 dw 194Bh           ; "signatures" used for Ext_BIOS
components
E000:61D0 dw 195Bh           ; decompression
E000:61D2 dw 194Bh           ; Init_mainboard_equipment & CPU
microcode
E000:61D2                         ; chk ISA CMOS chksum ?
E000:61D4      dw 1ABCh           ; Check checksum. Initialize
keyboard controller
E000:61D4                         ; and set up all of the 40: area
data.
E000:61D6      dw 1B08h           ; Relocate extended BIOS code
E000:61D6                         ; init CPU MTRR, PCI REGs(Video
BIOS ?)
E000:61D8      dw 1DC8h           ; Video_Init (including EPA proc)
E000:61DA      dw 2342h
E000:61DC      dw 234Eh           ; dummy
E000:61DE      dw 2353h           ; dummy
E000:61E0      dw 2355h           ; dummy
E000:61E2      dw 2357h           ; dummy
E000:61E4      dw 2359h           ; init Programmable Timer (PIT)
E000:61E6      dw 23A5h           ; init PIC_1 (programmable
Interrupt Ctrl)
E000:61E8      dw 23B6h           ; same as above ?
E000:61EA      dw 23F9h           ; dummy
E000:61EC      dw 23FBh           ; init PIC_2
E000:61EE      dw 2478h           ; dummy
E000:61F0      dw 247Ah           ; dummy
E000:61F2      dw 247Ah
E000:61F4      dw 247Ah
E000:61F6      dw 247Ah
E000:61F8      dw 247Ch           ; this will call RAM_POST_tests
again
E000:61FA      dw 0               ; for values below(a.k.a ISA POST)
E000:61FA      dw 0
E000:61FA       END_E0_POST_TESTS_TABLE

E000:247C last_E000_POST proc near
E000:247C      cli                ; Clear Interrupt Flag
E000:247D      mov word ptr [bp+156h], 0
E000:2483      mov cx, 30h ; '0'
E000:2486      mov dl, 61FCh       ; this addr contains 0000h
E000:2489      repeat_RAM_POST_tests: ; CODE XREF: last_E000_POST+10
E000:2489      call RAM_POST_tests; this call immediately return
E000:2489      ; since cs:[di]=0000h
E000:248C      jb repeat_RAM_POST_tests ; jmp if CF=1; not taken
E000:248E      mov cx, 30h ; '0'
E000:2491      mov dl, 61FCh       ; cs:[di] contains 249Ch
E000:2494      repeat_RAM_POST_tests_2: ; CODE XREF: last_E000_POST+1B
E000:2494      call RAM_POST_tests; this call should nvr return if
E000:2494      ; everything is ok
E000:2497      jb repeat_RAM_POST_tests_2 ; Jump if Below (CF=1)
E000:2499      jmp Halt_System      ;
E000:2499       last_E000_POST endp

E000:61FC       ISA_POST_TESTS
E000:61FC      dw 0
E000:61FE      dw 249Ch
E000:6200      dw 26AFh
E000:6202      dw 29D4h
E000:6204      dw 2A54h           ; dummy
E000:6206      dw 2A54h
E000:6208      dw 2A54h

E000:620A      dw 2A54h
E000:620C      dw 2A54h
E000:620E      dw 2A54h
E000:6210      dw 2A56h  ; dummy
E000:6212      dw 2A56h
E000:6214      dw 2A56h
E000:6216      dw 2A58h
E000:6218      dw 2A64h
E000:621A      dw 2B38h
E000:621C      dw 2B5Eh  ; dummy
E000:621E      dw 2B60h  ; dummy
E000:6220      dw 2B62h
E000:6222      dw 2B8Ch  ; HD init ?
E000:6224      dw 2BF0h  ; game io port init ?
E000:6226      dw 2BF5h  ; dummy
E000:6228      dw 2BF7h  ; FPU error interrupt related
E000:622A      dw 2C53h  ; dummy
E000:622C      dw 2C55h
E000:622E      dw 2C61h  ; dummy
E000:6230      dw 2C61h
E000:6232      dw 2C61h
E000:6234      dw 2C61h
E000:6236      dw 2C61h
E000:6238      dw 2C61h
E000:623A      dw 2C61h
E000:623C      dw 6294h  ; set cursor characteristic
E000:623E      dw 62EAh
E000:6240      dw 6329h
E000:6242      dw 6384h
E000:6244      dw 64D6h  ; dummy
E000:6246      dw 64D6h
E000:6248      dw 64D6h
E000:624A      dw 64D6h
E000:624C      dw 64D6h
E000:624E      dw 64D6h
E000:6250      dw 64D6h
E000:6252      dw 64D6h
E000:6254      dw 64D6h
E000:6256      dw 64D6h
E000:6258      dw 64D6h
E000:625A      dw 64D6h
E000:625C      dw 64D6h
E000:625E      dw 64D8h  ; bootstrap
E000:6260      dw 66A1h
E000:6262      dw 673Ch
E000:6264      dw 6841h  ; issues int 19h (bootstrap)
E000:6266      dw 0

E000:6266 END_Isa_POST_TESTS

Note:

- The "POST jump table" procedures will set the Carry Flag (CF=1) if they encounter something wrong during their execution. Upon returning of the POST procedure, the Carry Flag will be tested, if it's set, then the "RAM_POST_TESTS" will immediately returns which will Halt the machine and output sound from system speaker.
At the "segment vector" routine. Below is only an example of its usage. There are lot of places where it's implemented. There are couple of variation of this "segment vector". Some will jump from segment E000h to F000h, some will jump from segment F000h to E000h, some jump from E000h to 6000h(relocated decompressed awardext.rom) and some jump from F000h to 6000h(relocated decompressed awardext.rom).

First variant: jump from segment E000h to F000h

<table>
<thead>
<tr>
<th>Address</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>E000:1553</td>
<td>Restore_WarmBoot_Flag proc near ; CODE XREF:</td>
</tr>
<tr>
<td>E000:155A</td>
<td>call F000_read_cmos_byte ; Call Procedure</td>
</tr>
<tr>
<td>E000:156E</td>
<td>Restore_WarmBoot_Flag endp</td>
</tr>
</tbody>
</table>

Address    Machine Code          Assembly Code
E000:6CA2                       F000_read_cmos_byte proc near ; CODE XREF:  
E000:6CA2 68 00 E0               push 0E000h        ; CODE XREF:   
E000:6CA2 68 B3 EC               push 6CB3h          ; Read_CMOS_byte  
E000:6CA8 68 31 EC               push 0EC31h        ; Jump  
E000:6CAB 68 FD E4               push 0E4FDh       ; -----------------------------------  
E000:6CAE 008 EA 30 EC 00 F0         jmp far ptr F000_func_vector ;   
E000:6CB3 008 C3                     retn               ; Return Near  
F000:6CB3                       F000_read_cmos_byte endp ; sp = -8 |

Address    Machine Code          Assembly Code
F000:EC30                       F000_func_vector: ; CODE XREF: |
F000:EC30                        ; reinit_CPU?+12  
F000:EC30 C3 |
F000:EC31                        ; target function  
F000:EC31 008 C3 |
F000:EC31 CB |
F000:EC31 CB  |
F000:EC31 CB segment vector |

F000:E4FD                       read_CMOS_byte proc near ; CODE XREF: |
F000:E4FD |
F000:E4FD 87 DB xchg bx, bx ; Exchange  
Copyright © 2004 and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited.
Second variant: jump from segment E000h to 6000h

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembly Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>E000:171F</td>
<td></td>
<td>Check_F_Next proc near ; CODE XREF: chksum_ROM+2D</td>
</tr>
<tr>
<td>E000:1737</td>
<td>0E</td>
<td>push cs</td>
</tr>
<tr>
<td>E000:1738</td>
<td>68 43 17</td>
<td>push 1743h ; ret addr below</td>
</tr>
<tr>
<td>E000:173B</td>
<td>68 29 18</td>
<td>push 1829h ; func addr in XGroup seg (Detect EEPROM)</td>
</tr>
<tr>
<td>E000:173E</td>
<td>EA 02 00 00 60</td>
<td>jmp far ptr 6000h:2 ; jump to XGroup code</td>
</tr>
<tr>
<td>E000:1743</td>
<td></td>
<td>; sp = -6</td>
</tr>
</tbody>
</table>

----------------------------------------
| E000:1743 F8 | clc | ; Clear Carry |
| E000:1744 C3 | retn | ; Return Near |

----------------------------------------
| 6000:0000 | loret_6000_0: | ; CODE |
| 6000:0001 | C3 | ; jump to target procedure |
| 6000:0001 |              | ; sp = -6     |

----------------------------------------
| 6000:0000 | 68 01 00 | push 1 ; push return addr for retn |
| 6000:0002 |              | ; |
| 6000:0005 | 50 | push ax |
| 6000:0006 | 9C | pushf ; Push Flags Register onto the Stack |
| 6000:0007 | FA | cli ; Clear Interrupt Flag |
Third variant: jump from segment 6000h to F000h

Address        Assembly Code

6000:4F60 reinit_chipset proc far
6000:4F60      push ds
6000:4F61      mov ax, 0F000h
6000:4F64      mov ds, ax ; ds = F000h
6000:4F66      assume ds:nothing
6000:4F66      mov bx, 0E38h ; ptr to PCI reg vals (ds:bx = F000:E38h)
6000:4F69      next_PCI_reg: ; CODE XREF: reinit_chipset+3D
6000:4F69      cmp bx, 0EF5h ; are we finished ?
6000:4F6D      jz exit_PCI_init ; if yes, then exit
6000:4F6F      mov cx, [bx+1] ; cx = PCI addr to read
6000:4F72      call setup_read_write_PCI ; on ret, ax = F70Bh, di = F725h
6000:4F75      push cs
6000:4F76      push 4F7Fh
6000:4F79      push ax ; goto F000:F70B
(Read_PCI_Byte)
6000:4F7A      jmp far ptr 0E000h:6188h ; goto_seg_F000
6000:4F7F ; -------------------------------

6000:4F7F      mov dx, [bx+3] ; reverse-and mask

E000:6188      goto_F000_seg: ; CODE XREF: HD_init_?+3BD
E000:6188      ; HD_init_?+578
...
E000:6188 68 31 EC      push 0EC31h
E000:618B 50      push ax
At "chksum_ROM" procedure. This procedure is part of the "E0_POST_TESTS", which is the POST routine invoked using the "POST jump table". There's no immediate return from within this procedure. But, a call into "Check_F_Next" will accomplish the "near return" needed to proceed into the next "POST procedure" execution.
• The original tmp decompression routine for the "Extension_BIOS components" is one of the most confusing thing to comprehend at first. But, by understanding it, we "virtually" have no more thing to worry about the "BIOS code execution path". I suspect that the same technique as what I'm going to explain here is used across the majority of award bios. The basic run-down of this routine explained below.

1. **Expand_Bios** procedure called from the "main bootblock code execution path" saved the needed "signature" to the predefined area in RAM as shown below:

```
F000:E512 Expand_Bios proc near  ; CODE XREF: F000:E3DC
        .......... 
F000:E555  mov  bx, 0           ; mov bx,Temp_VGA_Seg
F000:E558  mov  es, bx          ; es = 0000h
F000:E55A  assume es:nothing
F000:E55A  mov  word ptr es:7004h, 0FFFFh ; mov word
             es:[Temp_VGA_Off+4],ffffh
F000:E55A  Ext_BIOS
F000:E55A   ........           ; later used for other
F000:E561  xor   al, al         ; clr expand flag
F000:E563  mov  bx, 1000h
F000:E566  mov  es, bx          ; es = 1000h; SrcSegment,i.e.
             seg_E000h
F000:E566  .......... 
F000:E568  assume es:nothing
F000:E568  xor bx, bx          ; bx = 0000h ; SrcOffset
F000:E56A  call  BootBlock_Expand ; read compressed
             original.tmp header and
             segment 5000h
F000:E56A  .......... 
             ecx=total_component_cmprssd_size
```

```
F000:E5B8 Expand_Bios endp
```
2. **Expand** procedure called from **Bootblock_Expand** procedure during Bootblock execution modify the header as needed and save the result in predefined area in RAM. The code as follows:

```
F000:7789 Expand proc near
........
F000:77FF    push    gs ; save gs
F000:7801    mov     di, 0 ; mov di, Temp_EXP_Seg
F000:7804    mov     gs, di ; gs = Temp_EXP_Seg (0000h)
F000:7806    assume    gs:nothing
F000:7806    mov     di, 6000h ; mov di, Temp_EXP_Off
F000:7809    mov     word ptr gs:[di], 7789h ; 0000:6000h = 7789h
F000:7809    ; mov word ptr gs:[di], offset Expand
F000:780E    add     bx, 12h ; bx = 12h
F000:7811    call    Get_Exp_Src_Byte ; get es:[bx+12h] to AL
(ExpSegment hi byte)
F000:7814    sub     bx, 12h ; restore bx value (first pass = 0000h)
F000:7817    cmp     al, 40h ; is "extension component" ?
F000:7817    (original.tmp)
F000:7817    cmp     al, 40h ; at 1st: al equ 50h
F000:7817    (awadext.rom)
F000:7817    ; at all other components: al equ 41h
F000:7817    ; The decompression caveat is here d00d !!!
F000:7819    jnz     Not_POST_USE ; jmp if no: for original.tmp and awadext.rom
F000:7819    jnz     Not_POST_USE ; goto decompress, otherwise no
F000:781B    add     bx, 11h ; bx = ExpSegment_lo_byte index
F000:781E    call    Get_Exp_Src_Byte ; al = ExpSegment_lo_byte
F000:7821    sub     bx, 11h ; restore bx
F000:7824    or     al, al ; segment 4000h ?
F000:7824    ; this is always 00h when Expand
F000:7824    ; called from within original.tmp
F000:7824    jnz     Record_to_buffer ; jmp if no
F000:7826    ; (all "extension component"
F000:7826    jnz     Record_to_buffer ; jmp if no
F000:7826    jnz     Record_to_buffer ; jmp if no
F000:7826    jnz     Record_to_buffer ; jmp if no
F000:7826    jnz     Record_to_buffer ; jmp if no
F000:7826    jnz     Record_to_buffer ; jmp if no
F000:7826    cmp     dword ptr gs:[di+4], 0 ; cmp dword [0000:6004]:0
F000:7828    ; 1st pass from original.tmp, (programmed by
F000:7828    ; 0000:6004]=FFFFh
F000:7828    ; Expand BIOS before jmp to original.tmp
F000:7828    ; jmp always taken from within original.tmp
F000:7830    F000:7830 Record_to_buffer: ; CODE XREF: Expand+9D
F000:7830    F000:7830 movzx    dx, al ; dx = ExpSegment_lo_byte
F000:7833    inc     bx ; bx = header_chksum_index
```

F000:7834 call Get_Exp_Src_Byte ; al = header_chksum
F000:7837 sub al, dl ; al = header_chksum -
ExpSegment_lo_byte
F000:7839 call Set_Exp_Src_Byte ; header_chksum = al
F000:783C dec bx ; restore bx
F000:783D xor al, al ; al = 00h
F000:783F add bx, 11h ; bx = ExpSegment_lo_byte
F000:7842 call Set_Exp_Src_Byte ; ExpSegment_lo_byte = 00h
(ExpSegment=4000h)
F000:7845 sub bx, 11h ; restore bx
F000:7848 inc dx ; dx = ExpSegment_lo_byte + 1
F000:7849 shl dx, 2 ; dx = 4*(ExpSegment_lo_byte + 1)
F000:784C add di, dx ; di = 6000h + dx (look above!)
F000:784E mov gs:[di], bx ; 0000:[di] = ComprssedCompnnt_offset_addr
F000:784E ; (offset addr in compressed Ext_BIOS)
F000:7851 mov cx, es ; cx = ExpSegment
F000:7853 mov gs:[di+2], cx ; 0000:[di+2]=ExpSegment
F000:7857 call Get_Exp_Src_Byte ; al = header_len
F000:785A movzx ecx, al ; ecx = header_len
F000:785E add bx, 7 ; bx --> point to compressed file size
F000:7861 call Get_Exp_Src_Dword ; eax = compressed file size
F000:7864 sub bx, 7 ; restore bx
F000:7867 add ecx, eax ; ecx = header_len + compressed_file_size
F000:786A add ecx, 3 ; ecx = total_compressed_component_size
F000:786E pop gs ; restore gs
F000:7870 assume gs:nothing
F000:7870 jmp exit_proc ; Jump
F000:7873 Not_POST_USE:
F000:7873 pop gs ; restore gs value
F000:7875 call MakeCRCTable ; initialize CRC-16 lookup table used later
F000:7878 call ReadHeader ; read compressed component header into
F000:7878 ; scratchpad @RAM, on error
F000:7878 CF=1
F000:7878
F000:7878 jb exit_proc ; error, something wrong (CF=1)
F000:787B mov ecx, ds:10Ah ; mov ax,ExpSegment
F000:787F mov ds:104h, ax ; mov TgtSegment,ax
F000:7882 mov ax, ds:10Ah ; mov ax,ExpOffset
F000:7885 mov ds:106h, ax ; mov TgtOffset,ax
F000:7888 mov ecx, ds:310h ; mov ecx,compsize ;compressed size
F000:788B xor eax, eax ; eax = 0000 0000h
F000:7890 mov al, ds:571Ch ; mov al,headersize;
compressed header size
F000:7896 add ecx, eax ; Add
F000:7899 add ecx, 3 ; add ecx,COMPRRESSED_UNKNOWN_BYTE;
F000:7899 ; ecx = "total compressed size"
F000:789D mov edx, ds:314h ; mov edx,origsize
F000:78A2 push edx
F000:78A4 push ecx ; save ecx (total compressed component size)
F000:78A6 push bx ; bx = 0000h
F000:78A7 add bx, 5 ; offset 5 ('-lh0-' or '-lh5-')
F000:78AA call Get_Exp_Src_Byte ; get compress or store type value
F000:78AD pop bx ; bx = 0000h (1st pass)
F000:78AE cmp al, '0' ; is it "-lh0-" ? first pass is no
F000:78B0 jnz Not_Store ; No, jump (first pass: jump taken)
F000:78B2 push ds
F000:78B3 push si
F000:78B4 push bx
F000:78B5 mov di, ds:10Ah ; mov di,ExpOffset
F000:78B9 movzx ax, byte ptr ds:571Ch ; movzx ax,byte ptr headersize
F000:78BE add ax, 2 ; ax = hdrsize + 2
F000:78C1 add bx, ax ; bx = hdrsize + 2 (assuming bx is 0000h)
F000:78C3 mov cx, ds:310h ; mov cx,word ptr compressed_size_lo_word
F000:78C7 mov ax, ds:108h ; mov ax,ExpSegment
F000:78CA mov es, ax ; es = ExpSegment
F000:78CC add cx, 3 ; cx = ceiling(compressed_size_lo_word)
F000:78CF shr cx, 2 ; transfer to dword unit
(compressed_size/4)
F000:78D2
F000:78D2 Get_Store_Data_Loop: ; CODE XREF: Expand+151
F000:78D2 call Get_Exp_Src_Dword ; read dword from compressed file in RAM
F000:78D5 add bx, 4 ; point to next dword
F000:78D8 stosd ; store in es:di
(ExpSegment:ExpOffset)
F000:78DA loop Get_Store_Data_Loop ; Loop while CX != 0
F000:78DC
F000:78DC pop bx ; bx = offset_after_cmprssed_filename
F000:78DD pop si
F000:78DE pop ds
F000:78DF jmp short Expand_Over ; Jump

F000:78E1 ; -----------------------------------------------

F000:78E1 Not_Store: ; CODE XREF: Expand+127
F000:78E1 push word ptr ds:104h ; push word ptr TgtSegment
F000:78E5 push word ptr ds:106h ; push word ptr TgtOffset
F000:78E9 push large dword ptr ds:314h ; push dword ptr origsize
F000:78EE ; extract content from compressed file
F000:78EE call Extract ; call LZH decompression routine
F000:78F1 pop dword ptr ds:314h ; pop dword ptr origsize
F000:78F6 pop word ptr ds:106h ; pop word ptr TgtOffset
F000:78FA pop word ptr ds:104h ; pop word ptr TgtSegment
The lines marked in blue color are the lines which are executed when this "decompression engine" is invoked from within original.tmp as in this nnoprom.bin decompression process. The lines marked with red color is where the "signature" are written into memory. For example, nnoprom.bin component is defined with ID: 4027h. This "component's handling" will arrive at Record_to_buffer where it's ID is processed. In this routine it's "index" will be saved. The index is calculated as follows (also look at the code above):

\[
\text{index} = 4 \times (\text{lo}_\text{byte} (ID) + 1)
\]

This index is used to calculate the address to save the information, in nnoprom.bin's case it is A0h (from \[4 \times (27h + 1)\]), so the address to save the information begins at 60A0h. As you can see above, the info first saved is the component's offset address within the compressed "Extension BIOS components", saved to address 60A0h, then the "expansion/decompression segment address" saved to 60A2h. This "expansion/ decompression segment address" always 4000h for all "extension BIOS components" as you can see in the code above. The same process is carried out for all other "extension BIOS components". I also have to note here that the source segment used for "extension BIOS components" decompression is 8000h this is due to the fact that Record_to_buffer in the Expand routine above only executed when called from Extern_execute2 routine as follows:

```
F000:C05B Extern_execute2 proc near ; CODE XREF: Expand_Bios+9B
    mov bx, 8000h ; mov bx,Temp_Extra_BIOS_Addres
F000:C05E mov es, bx ; es = 8000h
F000:C060 assume es:nothing
F000:C060 xor bx, bx ; bx = 0000h
F000:C062 xor ecx, ecx ; ecx= 0000 0000h
F000:C065 push cx ; assume no award external code
F000:C066
F000:C066 Expand_ROM_loop: ; CODE XREF: Extern_execute2+30
```

F000:C066  add  bx, cx          ; [bx] = next compressed
component
F000:C068  jb    Next_segment    ; Jump if Below (CF=1)
F000:C06A  test  ecx, 0FFFFFFFFh ; Logical Compare
F000:C071  jz    expand_awdext   ; Jump if Zero (ZF=1)
F000:C073
F000:C073  Next_segment:      ; CODE XREF:
Extern_execute2+D
F000:C073  mov   cx, es
F000:C075  add   cx, 1000h       ; Add
F000:C079  mov   es, cx          ; es = es + 1000h (next
segment)
F000:C07B  assume es:nothing
F000:C07B  jmp   short Expand_ROM_Next ; Jump
F000:C07D ;---------------------------------------------------
F000:C088  not_awdext_rom:   ; CODE XREF:
Extern_execute2+27
F000:C088  call  BootBlock_Expand ; on retn, cx =
total_comprssed_component_size
F000:C08B  jnb   Expand_ROM_loop ; Jump if Not Below (CF=0)
F000:C08D
F000:C08D  ----- decompress secondary extra BIOS area (0D000h)
-----
F000:C08D  mov   bx, es
F000:C08F  add   bx, 1000h       ; Add
F000:C093  mov   es, bx
F000:C095  assume es:nothing
F000:C097  xor   bx, bx          ; Logical Exclusive OR
F000:C09B  jnz   @@@F            ; No,skip
F000:C09F  pop   ax              ; restore flag
F000:C0A0  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   not_awdext_rom ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A7  jnz   @@@F            ; No,skip
F000:C0A9  pop   ax              ; restore flag
F000:C0AA  or    al, 1           ; set found flag
F000:C0A2  push  ax              ; store it to stack
F000:C0A4  cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
3. Next, the POST routine **POST_8S** a.k.a **Init Interrupt Vector** in original.tmp responsible for preparing the needed "signature" for the decompression as you can see below:

```assembly
E000:17B8 init_ivect proc near
........
E000:1834 ;for run time decompress code ret
E000:1834 mov bx, 2000h
E000:1837 mov es, bx
E000:1839 assume es:nothing
```
4. Next, init_NN-prom_BIN routine (this is just an example, other component will differ slightly) decompressed by the following code:

```
E000:71C1 init_NN-prom_BIN proc near ; CODE XREF: POST_13S
........
E000:71CF mov di, 0A0h ; 'a' ; di = offset_nnoprom.bin [nnoprom.bin-->4027h
E000:71CF ; di = 6000h + 4*(ExpSegment_lo_byte + 1) ;
E000:71CF ; A0h = 4h*(27h+1h) ]
E000:71CF ; look at Expand proc in bootblock
E000:71D2 call near ptr POST_decompress ; Call Procedure
E000:71D5 jb exit_proc ; jmp if CF=1, 1st pass CF=0
E000:71D9 push 4000h
E000:71DC pop ds ; ds = 4000h
E000:71DD assume ds:nothing
E000:71DD xor si, si ; si = 0000h
E000:71DF push 7000h
E000:71E2 pop es ; es = 7000h
E000:71E3 assume es:nothing
E000:71E3 xor di, di ; di = 0000h
E000:71E5 mov cx, 4000h
E000:71E8 cld ; Clear Direction Flag
E000:71E9 rep movsd ; move 64KB from seg_4000h to seg_7000h
E000:71E9 ; i.e. relocate decompressed code
```
E000:71EC  mov di, 3
E000:71EF  cmp dword ptr es:[di], 'ONN$' ; match nnprom.bin signature
E000:71F7  jnz exit_proc ; Jump if Not Zero (ZF=0)
E000:71FB  push 9FF8h
E000:71FE  pop es ; es = 9FF8h
E000:71FF  assume es:nothing
E000:7201  mov cx, 68h ; 'h'
E000:7204  xor al, al ; al = 0000h
E000:7206  rep stosb ; Store String
E000:7208  mov di, 0A4h ; 'a'
E000:720B  call near ptr POST_decompress ; Call Procedure
E000:720E  jb exit_proc ; Jump if Below (CF=1)
E000:7212  push ds
E000:7213  push es
E000:7214  push fs
E000:7216  push gs
E000:7218  call Update_Descriptor_Cache ; Call Procedure
E000:721D  xor esi, esi ; esi = 0000 0000h
E000:7220  mov ds, si ; ds = 0000h
E000:7222  assume ds:nothing
E000:7222  mov es, si ; es = 0000h
E000:7224  assume es:nothing
E000:7224  push 4000h
E000:7227  pop si ; si = 4000h
E000:7228  shl esi, 4 ; esi = 40000h
E000:722C  mov edi, 100000h
E000:7232  mov ecx, ebx
E000:7235  shr ecx, 2 ; Shift Logical Right
E000:7239  cld ; Clear Direction Flag
E000:723A  db 26h
E000:723A  rep movs dword ptr [edi], dword ptr [esi] ; Move Byte(s) from String to String
E000:723F  pop gs
E000:7241  pop fs
E000:7243  pop es
E000:7244  assume es:nothing
E000:7244  pop ds
E000:7245  assume ds:nothing
E000:7245  push 9FF8h
E000:7248  pop es
E000:7249  assume es:nothing
E000:7249  mov dword ptr es:0, 100000h
E000:7253  mov dword ptr es:4, 40000h
E000:725D  xor eax, eax ; Logical Exclusive OR
E000:7260  mov ax, 0E000h
E000:7263  shl eax, 4 ; Shift Logical Left
E000:7267  add eax, 7156h ; Add
E000:7267  mov es:8, eax
E000:7272  mov ax, 7
E000:7275  mov es:0Ch, ax
E000:7279  mov ax, 7000h
E000:727C  mov es:0Eh, ax
E000:7280  xor eax, eax ; Logical Exclusive OR
E000:7283  mov ax, 0E000h
E000:7286  shl eax, 4 ; Shift Logical Left
E000:728A  add eax, 71AAh ; Add
E000:7290  mov es:10h, eax
E000:7295  mov esi, 9FF80h
E000:729B  add esi, 0 ; Add
```assembly
e000:72a2 mov al, 36h ; '6'
e000:72a4 push cs
E000:72a5 push 72b0h
E000:72a8 push 0e4fdh ; read CMOS byte
E000:72ab jmp far ptr goto_F000_seg ; Jump
E000:72b0 ; ---------------------------------------------------
E000:72b0 mov bl, al
E000:72b2 mov ax, 0
E000:72b5 call near ptr init_nnoprom? ; Call Procedure
E000:72b8 pushf ; Push Flags Register onto
E000:72b9 the Stack
E000:72ba popf ; Pop Stack into Flags
E000:72bb mov ax, 0
E000:72bf or byte ptr ds:4b7h, 3 ; Logical Inclusive OR
E000:72c1 exit_proc: ; CODE XREF:
E000:72c1 init_NNOPROM_BIN+14 j
E000:72c2 ; init_NNOPROM_BIN+36 j ...
E000:72c6 popad ; Pop all General Registers
E000:72c7 (use32)
E000:72c8 pop es
E000:72c9 assume es:nothing
E000:72c9 pop ds
E000:72ca assume ds:nothing
E000:72ca retn ; Return Near from Procedure
E000:72ca init_NNOPROM_BIN endp ; sp =  6

E000:6e49 POST_decompress proc far ; CODE XREF:
EPA_Procedure+43
E000:6e49 ; EPA_Procedure+5E ...
E000:6e49 push ds
E000:6e4a push es
E000:6e4b push bp
E000:6e4c push di ; store DI
E000:6e4d push si ; store SI
E000:6e4e and di, 3ffe3h ; mask DI bit 14 and 15; 1st
call F000_Cpu_Cache ; enable caching
E000:6e52 cli ; Clear Interrupt Flag
E000:6e53 mov al, 0fFh ; mov al,TRUE
E000:6e55 push 0e0000h
E000:6e56 push 6e69h
E000:6e57 push 0e3c1h
E000:6e58 push 0e3d4h ; A20 On
E000:6e59 jmp far ptr F000 call ; turn on gate A20
E000:6e5a ; -------------------------------
E000:6e5b call E000_enter_FlatPMode ; Call Procedure
E000:6e5c mov ax, ds
E000:6e5d mov es, ax ; es = ds (flat 4GB addr
E000:6e5e space);
E000:6e5f assume es:nothing
E000:6e60 base_addr=0000 0000h
E000:6e61 call E000_back_to_RealMode ; restore ss
E000:6e62 pop dx ; dx = si
```
E000:6E74  pop ax             ; ax = di
E000:6E75  mov ebx, es:[di+6000h] ; mov ebx,es:[di+Temp_EXP_Offset]
E000:6E75                         ; ebx=0008[nnprom_cmPressed_offset]h (nnprom.bin)
E000:6E7B  or ebx, ebx        ; Logical Inclusive OR
E000:6E7E  cmp bx, 0FFFFh     ; Compare Two Operands
E000:6E89  test ah, 40h       ; 1st pass is 00h (ax = A0h)
E000:6E8C  cmp bx, 0FFFFh     ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6EB1  cmp ebx, 100000h   ; 1st pass this jmp IS taken
E000:6EB3  push di            ; save offset_Expand to stack
E000:6EB4  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6EB7  mov es, bx         ; es = ExpSegment (of the compressed component)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6EAA  cmp ebx, 100000h   ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6EB1  jb Is_New_Decompress_Method ; 1st pass this jmp IS taken
E000:6EB3  push di            ; save offset_Expand to stack
E000:6EB4  mov esi, ds:160000h ; mov esi = 90000h (last
E000:6EB7  mov es, bx         ; es = ExpSegment (of the compressed component)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6E92  mov di, es:6000h   ; di = offset_Expand
E000:6E92  ; offset addr saved by bootblock)
E000:6E97  mov esi, ds:160000h ; mov esi,[awardext.rom 4Byte hdr]
E000:6E9F  not esi            ; One's Complement Negation
E000:6EA2  cmp ds:80000h, esi  ; ExpSeg-CompOffset (ebx = 8xxxxh)
E000:6E92 Go_on:                  ; CODE XREF:
E000:6EFA mov es:[bx+11h], dx ; reset decompress segment
E000:6EFE add cl, ch ; original segment of checksum
E000:6F00 add dl, dh ; new segment of checksum
E000:6F02 sub cl, dh ; difference segment of checksum
E000:6F04 sub es:[bx+1], cl ; recalculate checksum
E000:6F08 jmp short decompress ; No skip process SI
E000:6F0A ; ----------------------------------------------
E000:6F0A Is_New_Decomp_Method: ; CODE XREF:
E000:6F0A POST_decompress+68
E000:6F0A add ebx, 0E0000h ; ebx = (80000h+E0000h) = 160000h
E000:6F11 mov cx, es:[ebx+11h] ; cx=ExpSegment(changed to 4000h by bootblock)
E000:6F16 push cx ; save ExpSegment
E000:6F17 push word ptr es:[ebx] ; save chksum and hdr_len
E000:6F1B test ah, 80h ; SI available? (1st pass no i.e. 00h)
E000:6F1E jz decompress ; 1st pass this jmp is taken
E000:6F20 mov es:[ebx+11h], dx
E000:6F25 add cl, ch ; Add
E000:6F27 add dl, dh ; Add
E000:6F29 sub cl, dh ; Integer Subtraction
E000:6F2B sub es:[ebx+1], cl ; Integer Subtraction
E000:6F30 E000:6F30 decompress: ; CODE XREF:
E000:6F30 decompress+AF
E000:6F30 push 6F49h ; goto decompression engine at seg_2000h
E000:6F33 mov dx, 3000h
E000:6F37 push 2000h
E000:6F3F push di
E000:6F48 retf ; jmp 2000:addr_of_Expand
E000:6F48 ; (goto decompression engine at seg_2000h)
E000:6F49 ; ----------------------------------------------
E000:6F49 push 0E000h
E000:6F4C push 6F5Ah
E000:6F4F push 0EC31h
E000:6F52 push 0E3D4h ; A20_On
E000:6F55 jmp far ptr F000_call ; jmp F000 A20_On
E000:6F5A ; ----------------------------------------------
E000:6F5A call E000_enter_PlatformMode ; Call Procedure
E000:6F5D mov ax, ds
E000:6F5F mov es, ax ; es-->BaseAddr=0000 0000h; limit 4GB
E000:6F61 assume es:nothing
E000:6F61 call E000_Back_to_RealMode ; Call Procedure
E000:6F64  mov eax, ds:80000h
E000:6F6B  cmp eax, ds:160000h ; 1st pass, ds:80000h equ
               (Not-dx:160000h)
E000:6F73  jnz Is_New_Decompress ; 1st pass this jmp is taken
E000:6F75  ror ebx, 10h ; Rotate Right
E000:6F79  mov es, bx
E000:6F7B  assume es:nothing
E000:6F7B  ror ebx, 10h ; Rotate Right
E000:6F7F  pop word ptr es:[bx]
E000:6F82  pop word ptr es:[bx+11h]
E000:6F86  mov ebx, es:[bx+0Bh]
E000:6F8B  jmp short disable_A20 ; Jump
E000:6F8D  ; -------------------------------------------
E000:6F8D

E000:6F8D Is_New_Decompress: ; CODE XREF:
POST_decompress+12A
E000:6F8D  pop word ptr es:[ebx] ; restore original
checksum
E000:6F91  pop word ptr es:[ebx+11h] ; restore original
segment
E000:6F96  mov ebx, es:[ebx+0Bh] ; get decompressed data
size
E000:6F9C
E000:6F9C disable_A20: ; CODE XREF:
POST_decompress+142
E000:6F9C  push 0E000h
E000:6F9F  push 6FADh
E000:6FA2  push 0EC31h
E000:6FA5  push 0E424h ; turn gate A20 off
E000:6FA8  jmp far ptr F000_call ; F000_CALL A20_Off
E000:6FAD  ; -------------------------------------------
E000:6FAD
clc ; Clear Carry Flag
E000:6FAE  jmp short POST_decomp_Ret ; Jump
E000:6FBD  ; -------------------------------------------
E000:6FBD

E000:6FBD

E000:6FBD Decompress_Data_Empty: ; CODE XREF:
POST_decompress+35
E000:6FBD  ; POST_decompress+3C
E000:6FBB  stc ; Set Carry Flag
E000:6FBD  ; POST_decompress+46
E000:6FBD  ; POST_decompress+165
E000:6FBD  pushf ; Push Flags Register onto
the Stack
E000:6FBD  push ebx
E000:6FBD  push 0E000h
E000:6FBD  push 6FC5h
E000:6FBD  push 0EC31h
E000:6FBD  push 0E3D4h ; turn on a20 gate
E000:6FBD  jmp far ptr F000_call ; F000_call A20_On
E000:6FBD  ; -------------------------------------------
E000:6FBD
call E000_enter_FlatPMode ; Call Procedure
E000:6FCE
E000:6FCE mov ax, ds
E000:6FCC  assume es:nothing
E000:6FCC  assume es:nothing
E000:6FCC  assume es:nothing

Copyright © 2004 and published by the CodeBreakers-Journal. Single print or electronic copies for personal use only are permitted. Reproduction and distribution without permission is prohibited.

what I’ve explained above only applies exactly to nnoprom.bin in my BIOS, but it's very possible that this mechanism still in use for other versions of award bios.

- After all of the explanation above, we only need to follow the "POST jump table execution" to be able to know which "execution path" is taken by the BIOS in which circumstances. Having doing this approach we'll be able to do what we please to our "to be hacked" award bios >:).

What I've explained above possibly far too premature to be ended here. But, I consider this article finished here as the Beta2 version of this article. If you follow this article from beginning to end, you'll absolutely be able to understand the "BIG Picture" of how the Award BIOS works. I think all of the issue dissected here is enough to do any type of modification you wish to do with award bios. If you find any mistake(s) within this article, please contact me. Goodluck with you BIOS reverse engineering journey, I hope you enjoy it as much as I do :) .