© The CodeBreakers-Journal, Vol.1, No.2. (2004)
http://www.CodeBreakers-Journal.com

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 throughly from beginning to end to get most out of it.

Contents

1. Foreword

I would like to welcome you to the darkside of a working example of spaghetty 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 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.

No.	Mnemonic (masm syntax)	Comment
1	pushad	save all the contents of General Purpose Registers
2	mov eax,80000064h	put the address of the PCI chip register to be accessed in eax (offset **64h** device 00:00:00 or hostbridge)
3	mov dx,0CF8h	put the address port in dx. Since this is PCI, we use **CF8h** as the port to open an access to the device.
4	out dx,eax	send the PCI address port to the I/O space of the processor
5	mov dx,0CFCh	put the data port in dx. Since this is PCI, we use **CFCh** as the data port to communicate with the device.
6	in eax,dx	put the data read from the device in eax
7	or eax, 00020202	modify the data (this is only example, don't try this in your machine, it may hang or even destroy your machine)
8	out dx,eax	send it back ....
9	.....	-
10	popad	pop all the saved register
11	ret	return...

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:

bit position	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
binary value	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	1	1	0	0	1	0	0
hexadecimal value	8				0				0				0				0				0				6				4			

- 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 , function 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 **295h** as its address port (a.k.a index port) and port **296h** as its data port. CMOS chip defines port **70h** as its address port and port **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 **backward compatiblity** 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.

Physical Address	**Used by**
000E 0000h - 000F FFFFh	1 Mbit, 2 MBit, and 4 MBit BIOSes
000C 0000h - 000D FFFFh	2 MBit, and 4 MBit BIOSes
0008 0000h - 000B FFFFh	4 MBit BIOSes

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

Space	Start	Size	Address Range	Comment
DOS	0	640K	00000000-0009FFFF	Cacheable
VGA	640K	128K	000A0000-000BFFFF	Used for SMM
BIOS	768K	16K	000C0000-000C3FFF	Shadow Ctrl 1
BIOS	784K	16K	000C4000-000C7FFF	Shadow Ctrl 1
BIOS	800K	16K	000C8000-000CBFFF	Shadow Ctrl 1
BIOS	816K	16K	000CC000-000CFFFF	Shadow Ctrl 1
BIOS	832K	16K	000D0000-000D3FFF	Shadow Ctrl 2
BIOS	848K	16K	000D4000-000D7FFF	Shadow Ctrl 2
BIOS	864K	16K	000D8000-000DBFFF	Shadow Ctrl 2
BIOS	880K	16K	000DC000-000DFFFF	Shadow Ctrl 2
BIOS	896K	64K	000E0000-000EFFFF	Shadow Ctrl 3
BIOS	960K	64K	000F0000-000FFFFF	Shadow Ctrl 3
Sys	1MB	–	00100000-DRAM Top	Can have hole
Bus	D Top		DRAM Top-FFFEFFFF	
Init	4G-64K	64K	FFFEFFFF-FFFFFFFF	000Fxxxx alias

The most important thing to take into account here is the address aliasing, as you can see the **FFFEFFFFh- FFFFFFFFh** 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. datasheet.

I/O Port address	Purpose
92h	Fast A20 and Init Register
4D0h	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)

I/O Port Locations	If U128E bit = 0	Function
70h and 74h	Also alias to 72h and 76h	Real-Time Clock (Standard RAM) Index Register
71h and 75h	Also alias to 73h and 77h	Real-Time Clock (Standard RAM) Target Register
72h and 76h		Extended RAM Index Register (if enabled)
73h and 77h		Extended RAM Target Register (if enabled)

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 147. RTC (Standard) RAM Bank (LPC I/F—D31:F0)

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

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 :

```
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 :

```
Address    Hex                  Mnemonic
F000:6000                       F000_6000_read_pci_byte proc near
F000:6000   66 B8 00 00 00 80   mov   eax, 80000000h
F000:6006   8B C1               mov   ax, cx          ; copy offset
addr to ax
F000:6008   24 FC               and   al, 0FCh        ; mask it
F000:600A   BA F8 0C            mov   dx, 0CF8h
F000:600D   66 EF               out   dx, eax
F000:600F   B2 FC               mov   dl, 0FCh
F000:6011   0A D1               or    dl, cl          ; get the byte
addr
F000:6013   EC                  in    al, dx          ; read the byte
F000:6014   C3                  retn                  ; Return Near
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                                                 ; base addr
= 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 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 PMode
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 :

```
Address    Hex                  Mnemonic
F000:E060 8C C8                 mov   ax, cs
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
> FFFF0000H. The starting address is thus formed by adding the base address to the value in the
> EIP register (that is, FFFF0000 + FFF0H = FFFFFFF0H).
> **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 **F 0000h** 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 tool 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 avalilable. 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 avalilable.
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 LZH 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 :

```
Address  Hex                                             ASCII
00000000 25F2 2D6C 6835 2D85 3A00 00C0 5700 0000 %.-lh5-.:...W...
00000010 0000 4120 010C 6177 6172 6465 7874 2E72 ..A ..awardext.r
00000020 6F6D DB74 2000 002C F88E FBDF DD23 49DB om.t ..,.....#I.
```

Beside 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:FFF0 (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 :** nnoprom.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 :

```
o     Address  Hex                                             ASCII
o     00037D00 2A42 4253 532A 0060 0070 0060 0060 00A0
*BBSS*.`.p.`.`..
o     00037D10 3377 4670 8977 ACCF C4CF 0100 00FF FFFF
3wFp.w.........
o     00037D20 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
................
o     00037D30 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
................
```

- 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 preceeding "-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 :

```
o     Address  Hex                                             ASCII
o     00003AA0 4E61 19E6 9775 2B46 BA55 85F0 0024 382D
Na...u+F.U...$8-
o     00003AB0 6C68 352D DC5C 0000 00A0 0000 0000 0140 lh5-
.\.........@
o     00003AC0 2001 0B43 5055 434F 4445 2E42 494E BCAA
..CPUCODE.BIN..
o     00003AD0 2000 0038 3894 9700 52C4 A2CF F040 0000
..88...R....@..
o     00003AE0 4000 0000 0000 0000 0000 0000 0000 0000
@...............
o     ........
o     000097A0 0E3C 8FA7 FFF4 FFFE 9FFF D3FF FFFB FF00
.&lt.............
o     000097B0 24D9 2D6C 6835 2DFA 0D00 00A6 2100 0000 $.-lh5-
.....!...
```

compressed ORIGINAL.TMP file in my BIOS :

```
Address  Hex                                             ASCII
0001FFF0 FFFF FFFF FFFF FFFF FFFF FFFF FFFF FFFF
................
00020000 251A 2D6C 6835 2D09 5501 0000 0002 0000 %.-lh5-
.U.......
```

```
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|...L
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 - F 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 explanantion 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 accross 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 FFFFFFF0H.
The answer is : I repeat that my northbridge chipset aliases address range **FFFE 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:FFF0** 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:

```
Address    Hex                      Mnemonic
F000:E07F BC 0B F8                  mov   sp, 0F80Bh      ; contains
E103h (memory presence test code)
F000:E082 E9 7B 15                  jmp   Ct_Very_Early_Init ; return
from this jump
F000:E082                                                 ; is
redirected to F000:E103h
```

At Reset PCI Bus routine:

```
Address    Hex                      Mnemonic
F000:E1A0 BF A6 E1                  mov   di, 0E1A6h      ; the return
addr of the jump below
F000:E1A3 E9 42 99                  jmp   Reset_PCI_Bus   ;
Jumpless_in_Decompress_Area,
F000:E1A3                                                 ; Program CPU
clock pin, host clock
F000:E1A3                                                 ; for
jumperless platform ???
....
```

```
F000:7CDD                          _delay:                 ; CODE XREF:
Reset_PCI_Bus+1F5
F000:7CDD E2 FE                      loop  _delay          ; Loop while
CX != 0
F000:7CDF FF E7                      jmp   di              ; jump back to
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]
(cs:[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 01 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
PMode flag
F000:E2B4 0F 22 C0                   mov   cr0, eax
F000:E2B7 EB 00                      jmp   short $+2       ; clear
prefetch, enter 16-bit PMode. We're
F000:E2B7                                                  ; using the
"unchanged" hidden value of CS
F000:E2B7                                                  ; register
(descriptor cache) from previous
F000:E2B7                                                  ; "PMode
session" in memory_check_routine
F000:E2B7                                                  ; for code
segment desc
F000:E2B9 B8 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
```

```
F000:E2C0                          ;There are two locations to access
E0000H ROM space, one is 0E0000H
F000:E2C0                          ;and another is 0FFFE0000H. Some
chipsets can not access onboard ROM
F000:E2C0                          ;space at 0E0000H if any device also
use the space on ISA bus. To
F000:E2C0                          ;solve this problem , we need to
change address to 0FFFE0000H
F000:E2C0                          ;to read BIOS contents at 0E0000H
space.
F000:E2C0                            assume es:nothing
F000:E2C0 66 BE 00 00 0E 00          mov   esi, 0E0000h    ; starting
addr of compressed original.tmp
F000:E2C6 67 66 81 7E 02 2D 6C 68+   cmp   dword ptr [esi+2], '5hl-' ;
LHA signature
F000:E2CF 74 07                      jz    LHA_sign_OK     ; Jump if Zero
(ZF=1)
F000:E2D1 66 81 CE 00 00 F0 FF       or    esi, 0FFF00000h ; esi =
FFFE0000h
F000:E2D8
F000:E2D8                          -- move entire BIOS (i.e. original.tmp
and bootblock)
F000:E2D8                          from ROM at E0000h-FFFFFh to RAM at
10000h-2FFFFh --
F000:E2D8
F000:E2D8                          LHA_sign_OK:            ; CODE XREF:
F000:E2CF
F000:E2D8 66 BF 00 00 01 00          mov   edi, 10000h     ; buffer at
1000:0
F000:E2DE 66 B9 00 80 00 00          mov   ecx, 8000h      ; copy 128
KByte to buffer (original.tmp &
F000:E2DE                                                  ; bootblock)
F000:E2E4 67 F3 66 A5                rep movs  dword ptr es:[edi], dword
ptr [esi] ; Move Bytes from
F000:E2E4
; String to String
F000:E2E8 0F 20 C0                   mov   eax, cr0
F000:E2EB 24 FE                      and   al, 0FEh        ; clear PMode
bit
F000:E2ED 0F 22 C0                   mov   cr0, eax
F000:E2F0 EB 00                      jmp   short $+2       ; clr
prefetch, back to RealMode
F000:E2F2 EA F7 E2 00 20             jmp   far ptr 2000h:0E2F7h ; Jump
below in RAM
F000:E2F7                          ; ---------------------------------
-----------------------------
F000:E2F7                          ;Setup temporary stack at 0:1000H, at
this point
F000:E2F7                          ;Bios code (last 128 Kbyte) is still
compressed
F000:E2F7                          ;except the bootblock and
decompression code
F000:E2F7
F000:E2F7                          BootBlock_in_RAM:       ; ax = 0000h
F000:E2F7 33 C0                      xor   ax, ax
F000:E2F9 8E D0                      mov   ss, ax          ; ss = 0000h
F000:E2FB                            assume ss:nothing
F000:E2FB BC 00 10                   mov   sp, 1000h       ; ss:sp =
0000:1000h
F000:E2FE
```

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:

```
Address    Hex                         Mnemonic
F000:FFF0 EA 5B E0 00 F0    jmp   far ptr entry_point ;
Northbridge is responsible for decoding
F000:FFF0                                          ; the target
address of this jump into BIOS
F000:FFF0                                          ; chip through
address aliasing. So, even if
F000:FFF0                                          ; this is a far
jump (read Intel Software
F000:FFF0                                          ; Developer
Guide Vol.3 for info)
F000:FFF0                                          ; we are still
in BIOS chip d00d ;)
F000:FFF0                                          ; vi693A:
FFFEFFFF-FFFFFFFF is 000Fxxxx alias.
```

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-EFFFFh** 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 - FFFFFh**) 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:

```
Address    Hex                          Mnemonic
F000:E3DC E8 33 01                      call  Expand_BIOS     ; decompress
bios code
F000:E3DF EB 03                         jmp   short BIOS_chksum_OK ;
checksum is good
F000:E3E1                             ; ---------------------------------
-----------------------------
F000:E3E1
F000:E3E1                             BIOS_chksum_err:        ; CODE XREF:
F000:E347
F000:E3E1                                                     ; F000:E350
...
F000:E3E1 B8 00 10                      mov   ax, 1000h
F000:E3E4
F000:E3E4                             BIOS_chksum_OK:         ; CODE XREF:
F000:E3DF
F000:E3E4 8E D8                         mov   ds, ax          ; ax = 5000H
if checksum ok
F000:E3E4                                                     ; setup source
for shadowing
F000:E3E4                                                     ; so, if ok,
ds = 5000h
F000:E3E6                               assume ds:nothing
F000:E3E6 B0 C5                         mov   al, 0C5h
F000:E3E8 E6 80                         out   80h, al         ;
manufacture's diagnostic checkpoint
F000:E3EA
F000:E3EA                             ;The source data segment is 5000H if
checksum is good.
F000:E3EA                             ;the contents in this area is
decompressed by routine "Expand_Bios".
F000:E3EA                             ;And segment 1000H is for shadowing
original BIOS image if checksum
F000:E3EA                             ;is bad. BIOS will shadow bootblock
and boot from it.
F000:E3EA E8 87 EB                      call  Shadow_BIOS_code ; Call
Procedure
F000:E3ED B0 00                         mov   al, 0           ; clear uP
cache
F000:E3EF E8 C7 10                      call  Enable_uP_cache ; Call
Procedure
F000:E3F2
F000:E3F2                             ;BIOS decide where to go from here.
F000:E3F2                             ;If BIOS checksum is good, this
address F80DH is shadowed by
F000:E3F2                             ;decompressed code (i.e. original.bin
and others),
F000:E3F2                             ;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-FFFDFFFFh** 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-FFFDFFFFh** in ROM chip into **8000:0000h - 9000:FFFFh** in DRAM.
4. Disable **FFF80000h-FFFDFFFFh** 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 - 2000:7FFDh**) and compare the result against result stored in **2000:7FFEh**. 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 origininal.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):

Offset from 1st byte	Offset in Real Header	Contents
00h	**N/A**	The header length of the component. It depends on the file/component name.
01h	**N/A**	The header 8-bit checksum, not including the first 2 bytes (header length and header checksum byte).

02h - 06h	**00h - 04h**	LZH Method ID (ASCII string signature). In my BIOS it's "-lh5-" which means: 8k sliding dictionary(max 256 bytes) + static Huffman + improved encoding of position and trees.
07h - 0Ah	**05h - 08h**	compressed file/component size in little endian dword value, i.e. MSB at **0Ah** and so forth
0Bh - 0Eh	**09h - 0Ch**	Uncompressed file/component size in little endian dword value, i.e. MSB at **0Eh** and so forth
0Fh - 10h	**0Dh - 0Eh**	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).
11h - 12h	**0Fh - 10h**	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).
13h	**11h**	File attribute. My BIOS components contain **20h** here, which is normally found in LZH level-1 compressed file.
14h	**12h**	Level. My BIOS components contain **01h** here, which means it's a LZH level-1 compressed file.
15h	**13h**	component filename name length in byte.
16h - [15h+filename_len]	**14h - [13h+filename_len]**	component filename (ASCII string)
[16h+filename_len] - [17h+filename_len]	**[14h+filename_len] - [15h+filename_len]**	file/component CRC-16 in little endian word value, i.e. MSB at **[HeaderSize - 2h]** and so forth.
[18h+filename_len]	**[16h+filename_len]**	Operating System ID. In my BIOS it's always **20h** (ASCII

		space character) which don't resemble any LZH OS ID known to me.
[19h+filename_len] - [1Ah+filename_len]	**[17h+filename_len] - [18h+filename_len]**	Next header size. In my BIOS it's always **0000h** which means no extension header.

- 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 decompresion process.

Location	Calculation Method
Right after compressed original.tmp	original.tmp 8-bit checksum. This value is calculated after it's copied to RAM at segment **1000h** and **2000h**. The code as follows : **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 cmprssd
F000:E31E                         ;
original.tmp in seg_F000h
F000:E321   adc   bx, 0           ; Add
with Carry
F000:E324   jz    below_or_equ_64Kb ; jmp
if compressed size less than 64Kb
F000:E326   mov   bx, cx          ; code
size remainder in next 64KB
F000:E326                         ;
(@seg_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   lodsb                 ; Load
String
F000:E32F   add   ah, al          ; calc
8 bit chksum, result in ah
F000:E331   loop  add_next_byte   ; loop
while cx != 0 (<64KB)
F000:E333
F000:E333   or    bx, bx          ;
compressed BIOS bigger than 64kb ?
F000:E335   jz    look_for_BBSS_sign ;
no, less than 64Kb
F000:E337   mov   cx, bx          ; cx =
compressed code size in next 64Kb
F000:E339   mov   bx, ds          ; setup
next 64Kb segment address
F000:E339                         ; at
first ds = 1000h
F000:E33B   add   bx, 1000h       ; next
64Kb
F000:E33F   mov   ds, bx          ;
ds=ds+1000h (ds = 2000h i.e. seg_F000h)
F000:E341   assume ds:nothing
F000:E341   xor   bx, bx          ; mark
that no next 64Kb ; bx = 0000h
F000:E343   jmp   short add_next_byte ;
continue to do checksum sum up
F000:E345 ; ---------------------------
---------------------------------
F000:E345
F000:E345 look_for_BBSS_sign:     ; CODE
```

	XREF: F000:E335 F000:E345 cmp ah, [si] ; cmp calc-ed chksum & chksum in image. F000:E345 ; in original.tmp BIOS image, F000:E345 ; chksum at 35531h (F_seg:5531h) F000:E347 jnz BIOS_cksm_error ; Jump if Not Zero (ZF=0)
Right after the decompression engine	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 ; the 4096th byte is the chksum F000:E364 ; at F000:7FFFh in my BIOS F000:E367 chksum_loop: ; CODE XREF: F000:E36A F000:E367 lodsb ; Load String F000:E368 add ah, al ; calc 8 bit chksum F000:E36A loop chksum_loop ; Loop while CX != 0 F000:E36C F000:E36C cmp ah, [si] ; decomp engine chksum OK ? F000:E36E jnz BIOS_cksm_error ; jump if no
1 byte before decompression engine checksum (that's explained above)	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
F000:E533 _8000h_chksum_done: ; CODE
XREF: Expand_Bios+18
F000:E533   mov   bx, 1000h  ; 1000h, 1st
64KB BIOS img (E000h seg of
F000:E533                     ; compressed
original.tmp)
F000:E536   mov   ds, bx      ; ds = 1000h
F000:E538   assume ds:nothing
F000:E538   xor   si, si      ; si = 0000h
F000:E53A   cld               ; Clear
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                     ; this is
the chksum for the bios binary
F000:E54F                     ; from
00000h to 37FFDh (C000:0h - F000:7FFDh)
F000:E551   jnz   BIOS_cksm_error ; Jump
if Not Zero (ZF=0)
```

The following are the key parts of the decompression routine :

Address Assembly Code

```
F000:E512 Expand_Bios proc near
.........
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   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
F000:E56A                         ; extract original.tmp
to segment 5000h
F000:E56A                         ; TgtSegment is read
from its LZH header
F000:E56A                         ; on return
ecx=total_component_cmprssd_size
F000:E56D   jb    decompression_error ; Jump if Below
(CF=1)
F000:E56F   test  ecx, 0FFFF0000h ; ecx & FFFF 0000h
;check against wrong
F000:E56F                         ; compressed
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:E5AB   jz    Expand_else     ; Jump if Zero (ZF=1)
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                          ; at 1st: al equ 50h
(original.tmp)
F000:7817                          ; at 2nd: al equ 41h
(awardext.rom)
F000:7817                          ; at all other
components: al equ 40h
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                          ; 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
```

```
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] =
CmprssedCompnnt_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
F000:7870   jmp   exit_proc       ; Jump
F000:7873 ; ---------------------------------------------
-----------------------------
F000:7873 Not_POST_USE:           ; CODE XREF: Expand+90
Expand+A5
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 CF=1
F000:7878                         ; bx preserved
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
;compressed size
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                         ; 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)
.........
F000:78E1 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
segmnt pointed by ds
F000:78FE                         ; i.e. clean up
scratch-pad RAM
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 :

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

---

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 **E0000h - FFFFFh** also in RAM. This is accomplished as follows:
    1. Reprogram the northbridge shadow RAM control register to enable **write only** into **E0000h - FFFFFh**, 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 **E0000h - FFFFFh**.
    3. Reprogram the northbridge shadow RAM control register to enable **read only** into **E0000h - FFFFFh**, 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 accross 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.

Address Range in RAM	Decompression State (by Bootblock code)	Description
0000:6000h - 0000:6xxxh	N/A	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**.
1000:0000h - 2000:5531h	Compressed	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.
2000:5532h - 2000:5FFFh	N/A	This area contains only padding bytes.
2000:6000h - 2000:FFFFh	Pure binary (executable)	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 "copy" of bootblock in segment **2000h**).
4100:0000h - 4100:57C0h	Decompressed	This area contains the decompressed awardext.rom. Note that the decompression process is accomplished by part of the bootblock in segment **2000h**.
5000:0000h - 6000:FFFFh	Decompressed	This area contains the decompressed original.tmp. Note that the decompression process is accomplished by part of the bootblock in segment **2000h**.
8000:0000h - 9000:FFFFh	Compressed	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**.
E000:0000h - F000:FFFFh	Decompressed	This area contains copy of the decompressed original.tmp, which is shadowed here by the bootblock in segment **2000h**.

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:

```
Address    Hex                          Mnemonic
F000:F80D                               This code is jumped into by the
bootblock code
F000:F80D                               if everything went OK
F000:F80D E9 02 F6                           jmp sysbios_entry_point ;
```

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 :

```
Address    Assembly Code
F000:EE12 sysbios_entry_point:    ; CODE XREF: F000:F80D
F000:EE12      mov ax, 0
F000:EE15      mov ss, ax         ; ss = 0000h
F000:EE17      mov sp, 1000h      ; setup stack at 0:1000h
F000:EE1A      call setup_stack   ; Call Procedure
F000:EE1D      call init_DRAM_shadowRW ; Call Procedure
F000:EE20      mov si, 5000h      ; ds=5000h (look at copy_mem_word)
F000:EE23      mov di, 0E000h     ; es=E000h (look at copy_mem_word)
F000:EE26      mov cx, 8000h      ; copy 64KByte
F000:EE29      call copy_mem_word ; copy E000h segment routine, i.e.
F000:EE29                         ; copy 64Kbyte from 5000:0h to
E000:0h
F000:EE2C      call j_init_DRAM_shadowR ; Call Procedure
F000:EE2F      mov si, 4100h      ; ds = XGroup segment decompressed,
i.e.
F000:EE2F                         ; at this point 4100h
F000:EE32      mov di, 6000h      ; es = new XGroup segment
F000:EE35      mov cx, 8000h      ; copy 64KByte
F000:EE38      call copy_mem_word ; copy XGroup segment , i.e.
F000:EE38                         ; 64Kbyte from 4100:0h to 6000:0h
F000:EE3B      call Enter_UnrealMode ; jump below in UnrealMode
F000:EE3E Begin_in_UnrealMode
F000:EE3E      mov ax, ds
```

```
F000:EE40      mov es, ax          ; es = ds (3rd entry in GDT)
F000:EE40                          ; base_addr=0000 0000h;limit 4GB
F000:EE42      assume es:nothing
F000:EE42      mov esi, 80000h     ; mov esi,(POST_Cmprssed_Temp_Seg
shl 4)
F000:EE42                          ; relocate lower 128KB bios code
F000:EE48      mov edi, 160000h
F000:EE4E      mov ecx, 8000h
F000:EE54      cld                 ; Clear Direction Flag
F000:EE55      rep movs dword ptr es:[edi], dword ptr [esi] ; move
F000:EE55                          ; 128k data to 160000h (phy addr)
F000:EE59      call Leave_UnrealMode ; Call Procedure
F000:EE59 End_in_UnrealMode
F000:EE5C      mov byte ptr [bp+214h], 0 ; mov byte ptr
F000:EE5C                          ; POST_SPEED[bp],Normal_Boot
F000:EE61      mov si, 626Bh       ; offset 626Bh (E000h POST tests)
F000:EE64      push 0E000h         ; segment E000h
F000:EE67      push si             ; next instruction offset (626Bh)
F000:EE68      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
Table Register
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                          ;
base_addr=F0000h;seg_limit=64KB;code,execute/ReadOnly
F000:5512                          ;
conforming,accessed;granularity=1Byte;16-bit segment;
F000:5512                          ; segment present,code,DPL=0
F000:551A      dq 8F93000000FFFFh ; data desc (10h)
F000:551A                          ; base_addr=0000
0000h;seg_limit=4GB;data,R/W,accessed;
F000:551A                          ; granularity=4KB;16-bit segment;
segment present,
F000:551A                          ; data,DPL=0
F000:5522      dq 0FF0093FF0000FFFFh ; data desc 18h
F000:5522                          ;
base_addr=FFFF0000h;seg_limit=64KB;data,R/W,accessed;
F000:5522                          ; 16-bit segment,granularity = 1
byte;
F000:5522                          ; segment present, data, DPL=0.
```

```
F000:552A      dq 0FF0093FF8000FFFFh ; data desc 20h
F000:552A                            ;
base_addr=FFFF8000h;seg_limit=64KB;data,R/W,accessed;
F000:552A                            ; 16-bit segment,granularity = 1
byte;
F000:552A                            ; segment present, data, DPL=0.
F000:5532      dq 930F0000FFFFh      ; data desc 28h
F000:5532                            ;
base_addr=F0000h;seg_limit=64KB;data,R/W,accessed;
F000:5532                            ; 16-bit segment,granularity = 1
byte;
F000:5532                            ; 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.

New Address Range in RAM	Decompression State	Description
6000:0000h - 6000:57C0h	Decompressed	This is the relocated awardext.rom
160000h - 17FFFFh	Compressed	This is the relocated compressed "BIOS extension", including the compressed awardext.rom. (i.e. this is the copy of **FFFC0000h - FFFDFFFF** in the BIOS rom chip.

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

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

```
E000:6276 ; --------------- S U B R O U T I N E ---------------------
-----------------
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
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      mov ax, cs:[di]     ; in the beginning :
E000:627F                          ; di = 61C2h ; ax = cs:[di] = 154Eh
E000:627F                          ; called from E000:2489 w/ di=61FCh
(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
E000:628E      jb 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
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                          ; on error generate spkr tone
E000:61CA      dw 1745h            ; Check CMOS circuitry
E000:61CC      dw 178Ah            ; "chipset defaults" initialization
E000:61CE      dw 1798h            ; init CPU cache (both Cyrix and
Intel)
E000:61D0      dw 17B8h            ; init interrupt vector, also
initialize
E000:61D0                          ; "signatures" used for Ext_BIOS
components
E000:61D0                          ; 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
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 Ctlr)
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
E000:61F8                       ; 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 di, 61FCh    ; this addr contains 0000h
E000:2489
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 di, 61FEh    ; cs:[di] contains 249Ch
E000:2494
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 29DAh
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 2BC8h            ; 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 2CA6h
E000:623C      dw 6294h            ; set cursor charcteristic
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**

```
Address    Assembly Code
```

```
E000:1553 Restore_WarmBoot_Flag proc near ; CODE XREF:
Restore_Boot_Flag
.........
E000:155A      call F000_read_cmos_byte ; Call Procedure
.........
E000:156E Restore_WarmBoot_Flag endp
```

```
Address    Machine Code          Assembly Code
E000:6CA2                        F000_read_cmos_byte proc near
E000:6CA2                                          ; CODE XREF:
Restore_WarmBoot_Flag+7
E000:6CA2                                          ;
sub_E000_1745+2
E000:6CA2 68 00 E0               push 0E000h
E000:6CA5 68 B3 6C               push 6CB3h
E000:6CA8 68 31 EC               push 0EC31h
E000:6CAB 68 FD E4               push 0E4FDh       ; Read_CMOS_byte
E000:6CAE 008 EA 30 EC 00 F0         jmp far ptr F000_func_vector ;
Jump
E000:6CB3                        ; ---------------------------------
------------------------
E000:6CB3 008 C3                     retn                ; Return Near
from Procedure
E000:6CB3                        F000_read_cmos_byte endp ; sp = -8
```

```
F000:EC30                        F000_func_vector:       ; CODE XREF:
sub_E000_1745+3C
F000:EC30                                          ;
reinit_CPU?+12
F000:EC30 C3                         retn                ; jump to
target function
F000:EC31                        ; ---------------------------------
------------------------
F000:EC31 CB                         retf                ; E000h
segment vector
```

```
F000:E4FD                        read_CMOS_byte proc near ; CODE XREF:
sub_F000_3CEE+1A
F000:E4FD                                          ;
sub_F000_3CEE+2A
F000:E4FD 87 DB                   xchg bx, bx        ; Exchange
Register/Memory with Register
```

```
F000:E4FF 90                    nop                 ; No Operation
F000:E500 E6 70                 out 70h, al         ; CMOS Memory:
F000:E500                                             ; used by
real-time clock
F000:E502 E3 00                 jcxz $+2            ; Jump if CX is 0
F000:E504 E3 00                 jcxz $+2            ; Jump if CX is 0
F000:E506 87 DB                 xchg bx, bx         ; Exchange
Register/Memory with Register
F000:E508 E4 71                 in al, 71h          ; CMOS Memory
F000:E50A E3 00                 jcxz $+2            ; Jump if CX is 0
F000:E50C E3 00                 jcxz $+2            ; Jump if CX is 0
F000:E50E C3                    retn                ; Return Near
from Procedure
F000:E50E                     read_CMOS_byte endp
```

Second variant: jump from segment **E000h** to **6000h**

```
Address    Machine Code          Assembly Code
```

```
E000:171F                   Check_F_Next proc near  ; CODE XREF:
chksum_ROM+2D
.........
E000:1737 0E                    push cs
E000:1738 68 43 17              push 1743h          ; ret addr below
E000:173B 68 29 18              push 1829h          ; func addr in
XGroup seg (Detect EEPROM)
E000:173E EA 02 00 00 60        jmp far ptr 6000h:2 ; jump to XGroup
code
E000:1743                   ; ---------------------------------------
---------------------------------
E000:1743 F8                    clc                 ; Clear Carry
Flag
E000:1744 C3                    retn                ; Return Near
from Procedure
E000:1744                   Check_F_Next endp ; sp = -6
```

```
6000:0000                     locret_6000_0:            ; CODE
XREF: 6000:0017
6000:0000 C3                        retn                ; jump
to target procedure
6000:0001                     ; ---------------------------------
---------------------------------
6000:0001 CB                        retf                ; back
to caller
6000:0002                     ; ---------------------------------
---------------------------------
```

```
6000:0002 68 01 00                  push 1              ; push
return addr for retn
6000:0002                                               ;
(addr_of retf above)
6000:0005 50                        push ax
6000:0006 9C                        pushf               ; Push
Flags Register onto the Stack
6000:0007 FA                        cli                 ; Clear
Interrupt Flag
```

```
6000:0008 87 EC                         xchg bp, sp             ;
Exchange Register/Memory with Register
6000:000A 8B 46 04                      mov ax, [bp+4]          ; mov
ax,1 ; look at 1st inst above
6000:000D 87 46 06                      xchg ax, [bp+6]         ; xchg
ax,word_pushed_by_org_tmp
6000:0010 89 46 04                      mov [bp+4], ax          ;
[sp+4] = word_pushed_by_org_tmp
6000:0013 87 EC                         xchg bp, sp             ;
modify sp
6000:0015 9D                            popf                    ; Pop
Stack into Flags Register
6000:0016 58                            pop ax
6000:0017 EB E7                         jmp short locret_6000_0 ; jump
into word_pushed_by_original.tmp
```

```
6000:1829 FA                            cli                     ; Clear
Interrupt Flag
.........
6000:18B3 C3                            retn                    ;
Return Near from Procedure
```

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
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
```

```
E000:618C 9C                    pushf               ; Push Flags
Register onto the Stack
E000:618D FA                    cli                 ; Clear Interrupt
Flag
E000:618E 87 EC                 xchg bp, sp         ; Exchange
Register/Memory with Register
E000:6190 8B 46 04              mov ax, [bp+4]      ; mov ax, EC31h
E000:6193 87 46 06              xchg ax, [bp+6]     ; xchg ret addr
and EC31h
E000:6196 89 46 04              mov [bp+4], ax      ; mov
[sp+4],[sp+6]
E000:6199 87 EC                 xchg bp, sp         ; Exchange
Register/Memory with Register
E000:619B 9D                    popf                ; Pop Stack into
Flags Register
E000:619C 58                    pop ax
E000:619D EA 30 EC 00 F0        jmp far ptr F000_func_vector ; Jump
```

```
F000:EC30                   F000_func_vector:       ; CODE XREF:
chk_cmos_circuit+3C
F000:EC30 C3                    retn                ; jump to target
function
F000:EC31                   ; ---------------------------------------
---------------------------
F000:EC31 CB                    retf                ; return to
calling segment:offset (6000:4F7F)
```

```
F000:F70B read_PCI_byte proc near ; CODE XREF: enable_ROM_write?+4
.........
F000:F724      retn               ; Return Near to F000:EC31h
F000:F724 read_PCI_byte endp
```

- 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.

```
E000:16D2                   chksum_ROM proc near
.........
E000:16FF 74 1E                 jz Check_F_Next     ; yes. This jump
will return this routine
E000:16FF                                           ; to where it's
called
.........
E000:171D EB E6                 jmp short spkr_endless_loop ; Jump
E000:171D                   chksum_ROM endp
```

```
E000:171F                   Check_F_Next proc near  ; CODE XREF:
chksum_ROM+2D
.........
E000:1743 F8                    clc                 ; signal
successful execution
```

```
E000:1744 C3                    retn                ; retn to
RAM_POST_TESTS, proceed to next POST proc
E000:1744                   Check_F_Next endp ; sp = -6
```

- 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 accross 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                         ; later used for other
Ext_BIOS
F000:E55A                         ; component decompression
F000:E561
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
F000:E56A                         ; extract original.tmp to
segment 5000h
F000:E56A                         ; on return
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                         ; at 1st: al equ 50h
(original.tmp)
F000:7817                         ; at 2nd: al equ 41h
(awdext.rom)
F000:7817                         ; at all other components: al
equ 40h
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                         ; 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
F000:7826   jnz   Record_to_buffer ; jmp if no
F000:7826                         ; (all "extension component"
jump here)
F000:7828   cmp   dword ptr gs:[di+4], 0 ; cmp dword
[0000:6004]:0
F000:7828                         ; 1st pass from original.tmp,
F000:7828                         ; [0000:6004]=FFFFh
(programmed by
F000:7828                         ; Expand BIOS before jmp to
original.tmp
F000:782E   jnz   Not_POST_USE    ; jmp always taken from
within original.tmp
F000:7830
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
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] =
CmprssedCompnnt_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
CF=1
F000:7878                          ; bx preserved
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
;compressed size
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                       ; 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
(cmprssd_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
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
F000:78FE Expand_Over:              ; CODE XREF: Expand+156
F000:78FE   call  ZeroFill_32K_mem ; zero fill 32K in segmnt
pointed by ds
F000:78FE                           ; i.e. clean up scratch-pad
RAM
F000:7901   pop   ecx               ; ecx = "total compressed
size" (restore ecx)
F000:7903   pop   edx
F000:7905   clc                     ; decompression success
F000:7906 exit_proc:                ;
F000:7906   pop   es
F000:7907   pop   bx
F000:7908   pop   eax
F000:790A   retn                    ; Return Near from Procedure
F000:790A Expand endp
```

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):
index = 4*(lo_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 * (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
F000:C05B   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, 0FFFF0000h ; 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:C07D
F000:C07D expand_awdext:          ; CODE XREF:
Extern_execute2+16
F000:C07D   cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C082   jnz   not_awdext_rom  ; No,skip
F000:C084   pop   ax              ; restore flag
F000:C085   or    al, 1           ; set found flag
F000:C087   push  ax              ; store it to stack
F000:C088
F000:C088 not_awdext_rom:         ; CODE XREF:
Extern_execute2+27
F000:C088   call  BootBlock_Expand ; on retn, cx =
total_comprssd_cmpnent_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:C095   xor   bx, bx          ; Logical Exclusive OR
F000:C097
F000:C097 Expand_ROM_Next:        ; CODE XREF:
Extern_execute2+20
F000:C097   xor   cx, cx          ; cx = 0000h
F000:C099
F000:C099 Expand_ROM_loop1:       ; CODE XREF:
Extern_execute2+4E
F000:C099   add   bx, cx          ; [bx] =
compressed_component_1st_byte
F000:C09B   cmp   byte ptr es:[bx+12h], 41h ; Is award external
code?
F000:C0A0   jnz   @@@F            ; No,skip
F000:C0A2   pop   ax              ; restore flag
F000:C0A3   or    al, 1           ; set found flag
F000:C0A5   push  ax              ; store it to stack
F000:C0A6
F000:C0A6 @@@F:                   ; CODE XREF:
Extern_execute2+45
F000:C0A6   call  BootBlock_Expand ; Call Procedure
F000:C0A9   jnb   Expand_ROM_loop1 ; Jump if Not Below (CF=0)
F000:C0AB   pop   ax
```

```
F000:C0AC   or    al, al          ; check award external code
has found?
F000:C0AE   retn                  ; Return Near from Procedure
F000:C0AE Extern_execute2 endp
```

```
F000:E5B9 BootBlock_Expand proc near ; CODE XREF:
Extern_execute2+2D
F000:E5B9                         ; Extern_execute2+4B ...
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); ax
=8000h(2nd pass)
F000:E5F6   shr   ax, 0Ch         ; ax = 1h
F000:E5F9   mov   es, ax          ; es = 1h(1st pass);es=8h(2nd
pass)
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 (1st pass); es =
8000h (2nd pass)
F000:E602   assume es:nothing
F000:E602   pop   ax
F000:E603   retn                  ; Return Near from Procedure
F000:E603 BootBlock_Expand endp
```

---

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 :

```
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
```

```
E000:1839      mov byte ptr es:0DFFFh, 0CBh ; 'T'
E000:183F      mov si, 0
E000:1842      mov ds, si          ; ds = 0000h
E000:1844      assume ds:nothing
E000:1844      mov si, 7000h
E000:1847      mov ax, [si+4]      ; ax = FFFFh (0000:7004h
filled before by
E000:1847                          ; Expand_Bios routine in
bootblock)
E000:184A      mov di, 0           ; es = 0000h
E000:184D      mov es, di
E000:184F      assume es:nothing
E000:184F      mov di, 6000h
E000:1852      mov es:[di+4], ax   ; [0000:6004] = FFFFh
E000:1856      cmp ax, 0FFFFh      ; Compare Two Operands
E000:1859      jz signature_ok     ; Jump if Zero (ZF=1)
E000:185B      mov ax, [si]
E000:185D      mov es:[di+4], ax
E000:1861      mov ax, [si+2]
E000:1864      shr ax, 0Ch         ; Shift Logical Right
E000:1867      mov es:[di+6], ax
E000:186B
E000:186B signature_ok:            ; CODE XREF: init_ivect+A1
E000:186B      call sub_E000_8510  ; Call Procedure
E000:186E      clc                 ; Clear Carry Flag
E000:186F      retn                ; Return Near from Procedure
E000:186F init_ivect endp
```

4. Next, init_NNOPROM_BIN routine (this is just an example, other component will differ slightly) decompressed by the following code :

```
E000:71C1 init_NNOPROM_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 for info
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
nnoprom.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
E000:71FF      xor di, di         ; di = 0000h
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
E000:726D      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
```

```
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
the Stack
E000:72B9      popf                ; Pop Stack into Flags
Register
E000:72BA      jb exit_proc        ; Jump if Below (CF=1)
E000:72BC      mov ax, 0
E000:72BF      mov ds, ax
E000:72C1      assume ds:nothing
E000:72C1      or byte ptr ds:4B7h, 3 ; Logical Inclusive OR
E000:72C6
E000:72C6 exit_proc:               ; CODE XREF:
init_NNOPROM_BIN+14 j
E000:72C6                          ; init_NNOPROM_BIN+36 j ...
E000:72C6      popad               ; Pop all General Registers
(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, 3FFFh       ; mask DI bit 14 and 15; 1st
pass di = A0h
E000:6E52      cli                 ; Clear Interrupt Flag
E000:6E53      mov al, 0FFh        ; mov al,TRUE
E000:6E55      call F000_Cpu_Cache ; enable caching
E000:6E58      push 0E000h
E000:6E5B      push 6E69h
E000:6E5E      push 0EC31h
E000:6E61      push 0E3D4h         ; A20_On
E000:6E64      jmp far ptr F000_call ; turn on gate A20
E000:6E69 ; ---------------------------------------------------
------------------------
E000:6E69      call E000_enter_FlatPMode ; Call Procedure
E000:6E6C      mov ax, ds
E000:6E6E      mov es, ax          ; es = ds (flat 4GB addr
space);
E000:6E6E                          ; base_addr=0000 0000h
E000:6E70      assume es:nothing
E000:6E70      call E000_Back_to_RealMode ; restore ss
E000:6E73      pop dx              ; dx = si
```

```
E000:6E74      pop ax              ; ax = di
E000:6E75      mov ebx, es:[di+6000h] ; mov
ebx,es:[di+Temp_EXP_Off]
E000:6E75                          ;
ebx=0008[nnoprom_cmpressd_offset]h (nnoprom.bin)
E000:6E7B      or ebx, ebx         ; Logical Inclusive OR
E000:6E7E      jz Decomp_Data_Empty ; Jump if Zero (ZF=1)
E000:6E82      cmp bx, 0FFFFh      ; Compare Two Operands
E000:6E85      jz Decomp_Data_Empty ; Jump if Zero (ZF=1)
E000:6E89      test ah, 40h        ; 1st pass is 00h (ax = A0h)
E000:6E8C      jz Go_on            ; 1st pass this jump is taken
E000:6E8E      clc                 ; Clear Carry Flag
E000:6E8F      jmp POST_decomp_Ret ; Jump
E000:6E92 ; ---------------------------------------------------
-----------------------
E000:6E92
E000:6E92 Go_on:                   ; CODE XREF:
POST_decompress+43
E000:6E92      mov di, es:6000h    ; di = offset_Expand
(decompression engine
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      mov ds:80000h, esi
E000:6EAA      cmp ebx, 100000h    ; ExpSeg-CompOffset (ebx =
8xxxxh)
E000:6EB1      jb Is_New_Decomp_Method ; 1st pass this jmp IS
taken
E000:6EB3      push di             ; save offset_Expand to stack
E000:6EB4      mov esi, 90000h     ; ds:[esi] = 90000h (last
64KB of Ext_BIOS)
E000:6EBA      mov edi, 140000h    ; es:[edi] = 140000h
E000:6EC0      mov ecx, 4000h      ; copy last 64 KB of Ext_BIOS
to 140000h - 14FFFFh
E000:6EC6      cld                 ; Clear Direction Flag
E000:6EC7      rep movs dword ptr es:[edi], dword ptr [esi] ;
Move Byte(s) from String to String
E000:6ECB      mov esi, 160000h    ; ds:[esi] = addr_of_last
Ext_BIOS (128KB)
E000:6ED1      mov edi, 80000h     ; es:[edi] = target addr
E000:6ED7      mov ecx, 8000h      ; copy 128KB from 160000h-
17FFFFh to 80000h-9FFFFh
E000:6EDD      cld                 ; Clear Direction Flag
E000:6EDE      rep movs dword ptr es:[edi], dword ptr [esi] ;
Move Byte(s) from String to String
E000:6EE2      pop di              ; di = offset_Expand
E000:6EE3      ror ebx, 10h        ; Rotate Right
E000:6EE7      mov es, bx          ; es = ExpSegment (of the
compressed component)
E000:6EE9      assume es:nothing
E000:6EE9      ror ebx, 10h        ; restore ebx
E000:6EED      mov cx, es:[bx+11h] ; store decompress_segment
for
E000:6EED                          ; checksum recalculation
E000:6EF1      push cx             ; store it to stack
E000:6EF2      push word ptr es:[bx] ; store original checksum
value
E000:6EF5      test ah, 80h        ; test SI is available?
E000:6EF8      jz decompress       ; 1st pass this jmp is taken
```

```
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
E000:6F0A Is_New_Decomp_Method:    ; CODE XREF:
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:
POST_decompress+AF
E000:6F30                          ; POST_decompress+BF ...
E000:6F30      ror ebx, 10h        ; Rotate Right
E000:6F34      mov es, bx          ; es = SrcSegment (16h i.e.
160000h_linear_addr)
E000:6F36      ror ebx, 10h        ; restore ebx(ebx = 16xxxxh ;
E000:6F36                          ; 1st pass: xxxx-> cmpressed
nnoprom offset)
E000:6F3A      push cs             ; save current code segment
E000:6F3B      push 6F49h          ; ret addr below
E000:6F3E      push 0DFFFh
E000:6F41      mov dx, 3000h
E000:6F44      push 2000h
E000:6F47      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_FlatPMode ; 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_Decomp  ; 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_Decomp:          ; 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
E000:6FB0 ; ---------------------------------------------------
------------------------
E000:6FB0
E000:6FB0 Decomp_Data_Empty:      ; CODE XREF:
POST_decompress+35
E000:6FB0                         ; POST_decompress+3C
E000:6FB0      stc                ; Set Carry Flag
E000:6FB1
E000:6FB1 POST_decomp_Ret:        ; CODE XREF:
POST_decompress+46
E000:6FB1                         ; POST_decompress+165
E000:6FB1      pushf              ; Push Flags Register onto
the Stack
E000:6FB2      push ebx
E000:6FB4      push 0E000h
E000:6FB7      push 6FC5h
E000:6FBA      push 0EC31h
E000:6FBD      push 0E3D4h        ; turn on a20 gate
E000:6FC0      jmp far ptr F000_call ; F000_call   A20_On
E000:6FC5 ; ---------------------------------------------------
------------------------
E000:6FC5      call E000_enter_FlatPMode ; Call Procedure
E000:6FC8      mov ax, ds
E000:6FCA      mov es, ax         ; es = 4GB segment,
base_addr=0000 0000h
E000:6FCC      assume es:nothing
```

```
E000:6FCC      call E000_Back_to_RealMode ; Call Procedure
E000:6FCF      mov eax, ds:80000h
E000:6FD6      cmp eax, ds:160000h ; Compare Two Operands
E000:6FDE      jnz Not_Old_Decomp_Method ; 1st pass this jmp is
taken
E000:6FE0      mov edi, 80000h
E000:6FE6      mov ecx, 4000h
E000:6FEC      xor eax, eax       ; Logical Exclusive OR
E000:6FEF      cld                ; Clear Direction Flag
E000:6FF0      rep stos dword ptr es:[edi] ; clear 80000h to
8FFFFh
E000:6FF4      mov esi, 140000h
E000:6FFA      mov edi, 90000h
E000:7000      mov ecx, 4000h
E000:7006      cld                ; Clear Direction Flag
E000:7007      rep movs dword ptr es:[edi], dword ptr [esi] ;
Move Byte(s) from String to String
E000:700B
E000:700B Not_Old_Decomp_Method:  ; CODE XREF:
POST_decompress+195
E000:700B      push 0E000h
E000:700E      push 701Ch
E000:7011      push 0EC31h
E000:7014      push 0E424h        ; turn gate A20 off
E000:7017      jmp far ptr F000_call ; F000_CALL A20_Off
E000:701C ; ---------------------------------------------------
-----------------------
E000:701C      pop ebx
E000:701E      mov al, 0          ; mov al,FALSE
E000:7020      call F000_Cpu_Cache ; disable CPU cache
E000:7023      popf               ; Pop Stack into Flags
Register
E000:7024      pop bp
E000:7025      pop es
E000:7026      assume es:nothing
E000:7026      pop ds
E000:7027      retn               ; Return Near from Procedure
E000:7027 POST_decompress endp ; sp = -18h
```

---

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 :) .