Introduction to UEFI Technology

Abd El-Aziz, Mostafa
iocoder@aol.com

Tarek, Aly
aliiitarek@gmail.com

Eldefrawy, Amr
amr.f.eldfrawy@gmail.com

December 3, 2013

Contents

1 BIOS

1.1 Introduction to System BIOS

In the wonderful world of computing, it is well known that the operating system is an essential component of the computer system. Not only does the OS drive the computer hardware and control software programs, but it also provides complicated services and routines that can be used by programs to facilitate their job.

Most computers today are based on Von Neumenn architecture. This implies that a program should be first loaded to some memory, which is directly accessible by the processing unit, before execution. The operating system, like any other program, abides by that rule. We note that an

operating system is not a single piece of code; large operating systems (like openBSD and IBM AIX) consist of hundreds of programs.

In modern computer systems, volatile memory units (like static and dynamic memory) are used to store program code and data during execution. When the computer starts up, it is obvious that the RAM doesn't contain any program to be processed by the CPU. That's why firmware is a necessary component of the system.

The word "firmware" refers to a presistent computer memory that contains a program that is never lost (even if the power is off). This program is executed when the computer starts up, and it is responsible for loading the OS into the RAM. The word "firmware" is used to refer to the combination of this program and the memory in which it is stored. Usually, ROMs are used to store the firmware. The firmware is not only responsible for loading the OS; it provides many other services that we are going to discuss in this article.

The firmware of our personal computers is usually called BIOS, which stands for Basic Input/Output System. When the PC powered up, BIOS initializes and tests system hardware, then loads the operating system from the hard-disk. You usually see the messages that are printed by BIOS every time the PC boots up.

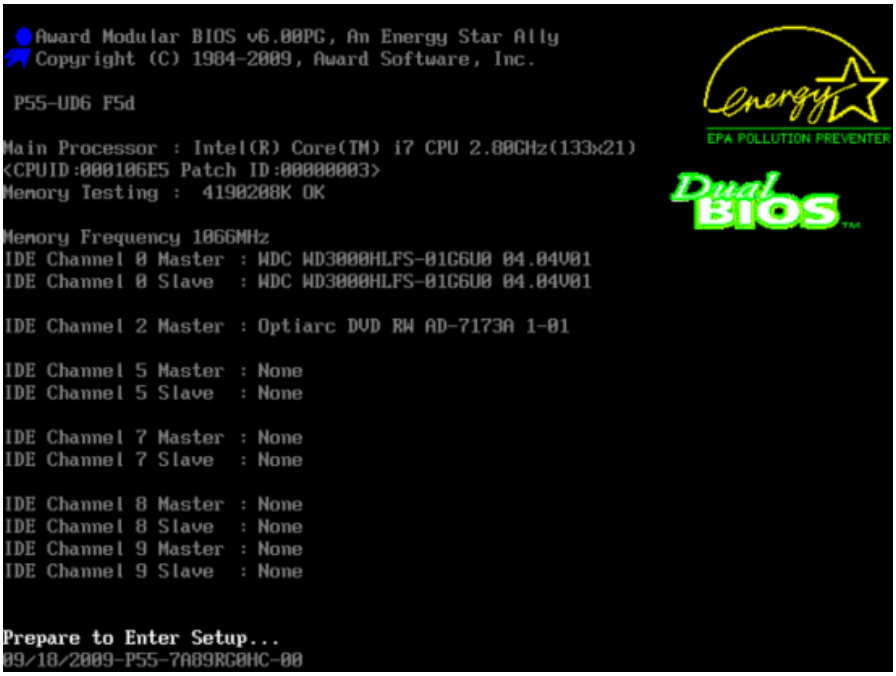

Figure 1: PC BIOS boot screen (Award).

In this article, we are going to discuss the operation of PC BIOS and emphasize its disadvantages, then introduce the new generation of PC firmware, which is called "Unified Extensible Firmware Interface (UEFI)".

1.1.1 BIOS Operations

As we have said in the introduction, BIOS does many tasks to do other than just loading the operating system. In the following list, we summarise the operations that are handled and maintained by BIOS:

1. A power-on self-test (PoST) for all hardware components is performed by BIOS immediately after the PC starts up. PoST aims to make sure that all components are working properly.

2. After PoST, the BIOS initializes the main hardware components of the system. For example, it initializes the keyboard controller and enables it to respond to keyboard signals.

3. The BIOS also provides a configuration interface that is usually called "CMOS/BIOS Setup Utility". It allows the user to change and configure BIOS settings.

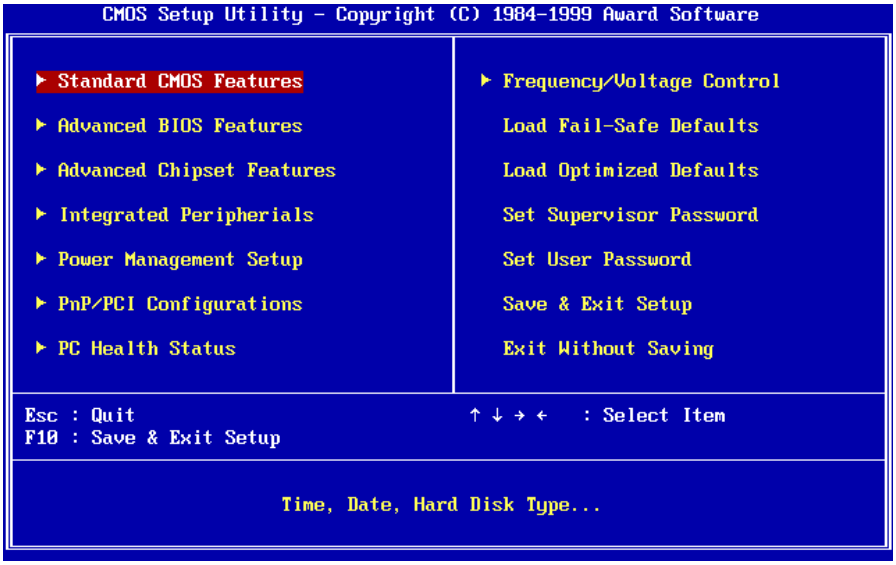

Figure 2: CMOS/BIOS Setup Utility.

4. As we described in the intro, the BIOS program loads the operating system after it has initialized the hardware.

5. The BIOS job doesn't stop here. In fact, after the OS is loaded, the BIOS acts as an interface between the hardware and the operating system. That's why we usually consider the system BIOS as the lowest-level of computer software (the level that direclty deals with the hardware).

1.1.2 BIOS Service Routines

One of the most important uses of BIOS is that it provides a set of routines that support the running operating system and programs. The basic idea is that x86 family supports the so called "software interrupts". Software interrupts are just like hardware interrupts, but the former are triggered by software programs (not by I/O devices).

Interrupt Number	Description
0x10	Video services
0x12	Memory services
0x13	Disk services
0x14	Serial port services
0x16	Keyboard services
0x17	Printer services

Table 1: Mostly used BIOS interrupt calls [2]

In x86 processors, hardware and software interrupts share the same interrupt vector table (IVT). Some of the interrupts are mapped to BIOS services routines. For example, interrupt 0x16 is attached to keyboard service routine, so when the program needs to call the subroutine that manages the keyboard, it shall send to the CPU an interrupt 0x16 signal. Table 1 shows some popular BIOS interrupts. [1]

1.2 BIOS Limitations

Through years and years, PC BIOS became full of troubles and limitations that make it difficult to develop complicated operating systems and systems software, and even to attach more advanced hardware. The lack for a definite standard for BIOS made it difficult for programmers to write standard code that properly works on all personal computers. BIOS works in real mode, while most modern operating systems work in protected and long modes. This complicates the interface between the OS and BIOS.

To illustrate those limitations, we show the boot procedure of an open-source operating system (Quafios) that mainly works in paged 32-bit mode. Because Quafios works in protected mode, it can't make use of BIOS routines during run-time. However, the OS needs to call some routines on start-up. Therefore, all necessary BIOS calls are handled before switching into protected mode. [3]

let's look at the main() function of the boot-loader:

```
int main() {

        // Enter Unreal Mode:
        enter_unreal();

        // Print splash:
        printf("\nQuafios boot loader is starting...\n");

        // Enable A20 Line:
        enable_a20();

        // Initialize bootinfo:
        bootinfo_init();

        // Decompress the ram disk:
```

```
        gunzip("RAMDISK.GZ");

        // Load kernel to memory:
        diskfs_load("/boot/kernel.bin", bootinfo->res[BI_KERNEL].base);

        // Enter protected mode & execute kernel.bin:
        go_protected(bootinfo->res[BI_KERNEL].base);

        // Set video mode 0x03:
        video_mode(0x03);

        // Kernel returned (this should never happen):
        printf("Quafios kernel main() returned!\n");

        // Done:
        return 0;
}
```

This C function performs a number of steps in order to boot the kernel:

1. To get around some of the BIOS limitations, Quafios uses a tricky trick of 386 architecture that allows the CPU to work in 16-bit mode with the ability to access memory above 1MB. This mode is called 'unreal mode'.
2. A special-purpose printf() function is written from scratch for the boot-loader, because BIOS support for console is very primitive.
3. Because BIOS doesn't support the OS with any means for decompression, gunzip() was written from scratch.
4. The boot-loader contains some redundant code that is compiled for 16-bit mode, in order to load the kernel (from the Live Disk) into extended memory.
5. go_protected() switches to protected mode and immediately gives control to the kernel that loads the remaining OS and never calls BIOS anymore.

As you can see, there are a lot of steps that are redundant between all operating systems, and because BIOS has many limitations, the code of the boot-loader is usually complicated. Therefore, we need more up-to-date firmware that doesn't have those limitations. That's why UEFI was introduced.

2 UEFI

2.1 Introduction to Unified Extensible Firmware Interface

The Unified Extensible Firmware Interface (UEFI) is an interface between the operating system and the firmware. It was preceded by the Extensible Firmware Interface (EFI) and therefore has a lot of its designations and protocols in its specification, but added a lot of functionality, flexibility and usability.

UEFI is of the following form:

1. Tables of data containing platform information.

2. booting service calls.

3. runtime service calls that help the OS load operate or boot.

The UEFI is provided to the OS loader to provide it with the protocols and methods it needs to boot the OS and also gives them to the OS.

The specifications of the UEFI are the set of structures and interfaces that the OS-Loader might use in booting. It is used to define the interfaces and structures that must be implemented by the firmware. Similarly with the OS, the developer must implement certain structures. Both the firmware and the OS developers can decide which of them to write these interfaces and structures.

Therefore, this interface is a way for communication between the OS and the firmware to support the Booting process and is accomplished by a complete and formal abstract specification of the software-visible interface given to the OS from the platform firmware. This in turn led to that an Operating System that used to run only on certain processor capabilities such as (16-bit or 32- bit) can now run on many types without any problems and without changing any code from the OS or the design and structure of the operating system and this is simply because the UEFI does this linkage for the OS and links it to the firmware.

This specification enables the replacement of legacy devices and firmware code by new associated codes and device types that are capable of providing the same exact functionality of these latter legacy devices and all of this is also done without any change in the OS. [4]

The UEFI was designed to be flexible and extendible. This appeared directly by allowing the manufacturer to add new functions that will further benefit their system and improve its performance. Its also applicable to a large spectrum of hardware platforms such as servers and mobile systems.

2.2 UEFI Components

The UEFI standard specification defines a number of components that any UEFI-compliant firmware should consist of. In this section, we are going to discuss the most important components of UEFI.

2.2.1 Boot Manager

UEFI allows dynamic loading of applications/drivers that are written specifically for UEFI. A program that is written for UEFI is called (UEFI image) and should be stored in PE32+ format which is an extension for PE32 format used by Microsoft Windows. UEFI images can be loaded by the boot- manager during run-time. This allows the firmware to be extensible. When an image is loaded, UEFI allocates an amount of memory for the image and then loads the program into the allocated memory and runs it.

Three types of UEFI images exist: [4]

1. UEFI applications: UEFI applications add usability value to UEFI, they are just like the applications that you use on Windows and Linux. But UEFI applications are written to

be used on UEFI. The firmware shall contain ready to use programs, like (ls, dir, ...). The firmware also contains a very important application called (UEFI shell) that can be executed on start-up and is used to load and execute other applications (just like UNIX shell). Figure 3 shows how to invoke (time) and (echo) commands from UEFI shell.

Figure 3: UEFI shell executing time and echo commands.

2. UEFI OS loader: A special type of UEFI applications that is used to load the operating system. When UEFI initialization is done, UEFI loads the UEFI OS loader (which is written by the OS developers and is stored on the disk). The OS loader is repsonsible for loading and giving control to the operating system. By default, when UEFI starts up, it looks for a usable UEFI OS loader on the boot disk. If it doesn't find anyone, it starts the UEFI shell to let the user invoke it manually (This should rarely happen). Notice that the user should have the option to force UEFI shell to start without looking for UEFI OS loaders. Figure 4 shows a screen-shot for a boot failure that causes UEFI shell to run automatically.

3. UEFI Drivers: A special type of UEFI images. They are used as device drivers for the firmware itself. By default, The firmware should contain built-in drivers that are loaded when the computer starts up, however, UEFI allows more drivers to be loaded during run-time and this provides maximum extensibility.

In the supplementary materials, we show how to write a UEFI application for UEFI and load it from UEFI shell.

2.2.2 UEFI Services

UEFI provides a number of services and functionalities to the OS loader and the OS itself. These services are present in any compliant system and are defined by interface functions that can be used in UEFI environment. Such environment includes: (1) protocols that control which devices can be accessed and when, (2) extended platform capabilities, and (3) applications running in the pre-boot environment and the OS loaders.

UEFI supports two types of these services:

1. Boot services
2. Runtime services.

Boot services are only available while the firmware owns the platform, while runtime services can still be used and accessed as the operating system is running.

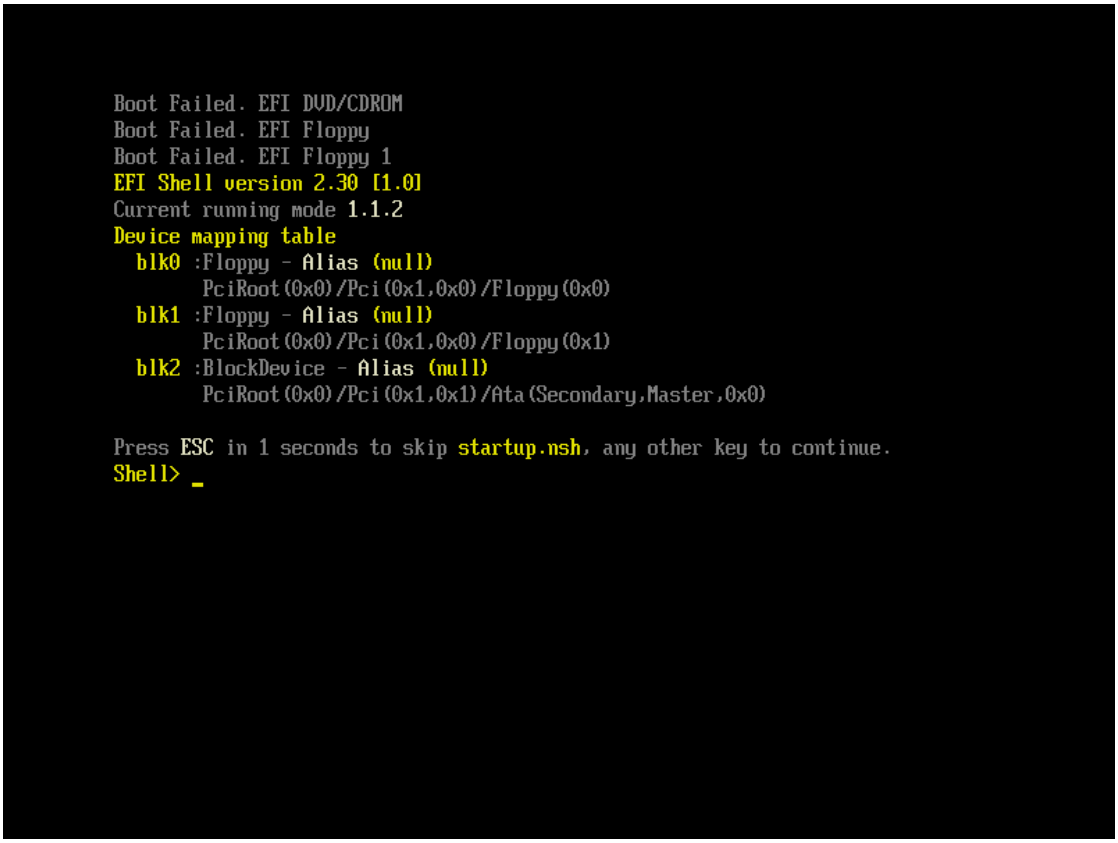

Figure 4: UEFI boot failure that automatically runs the shell.

Boot services include graphical consoles on many devices together with text consoles, and bus, block and file services. Runtime services include getting access to the date, time and NVRAM which stores non-volatile data.

Boot services are accessed before a successful call to "ExitBootServices()". Runtime services are available before and after a successful call to "ExitBootServices ()".

1. Variable Services: UEFI contain variables that provide a methodology to store non-volatile data, which is shared between platform firmware and operating systems or even UEFI applications. Variables are key/value pairs that can be used to keep crash messages in NVRAM after any crash for the operating system to retrieve after a reboot.

2. Time Services: UEFI provides device-independent time services. Time services include support for time zone and daylight saving fields, which allow the hardwares real time clock to be set to UTC or local time. On machines using a PC-AT real-time clock, the clock still has to be set to local time for compatibility with BIOS-based Windows.

During boot time, the system resources and functionalities are controlled and owned by the firmware and is accessed through boot services interface functions. The UEFI applications and the OS loader must use these boot services functions to be able to reach and access I/O devices

and allocate all the memory needed. All boot services functionality are available until the OS loader has been able to load its environment and can be able to take control over the system's operations.

2.3 UEFI vs BIOS

In the previous section we have discussed the major limitations of BIOS. The UEFI provides very good solutions for these limitations, and that's why it is much better to migrate from legacy BIOS to UEFI. In this sub-section, we discuss how UEFI solves the limitations of BIOS that we have already discussed.

2.3.1 UEFI is properly standardized

The original EFI specification was standardized by Intel. This EFI specification was the core on which the modern UEFI specs was built. Currently, UEFI is managed and standardized by "Unified EFI Forum". The UEFI Forum is an alliance of several companies like: Intel, HP, Microsoft, AMD, American Megatrends, and Phoenix Technologies. [5]

All computers that contain UEFI-compliant firmware should precisely conform to the standard specification documents that are released by UEFI forum. Therefore, if your program relies on some feature that is adopted by the UEFI specs, it is guaranteed that the program will run on all computers that conform to the same (or later) version of this specs.

2.3.2 UEFI can work in either 32-bit or 64-bit mode

If the computer is built on a 32-bit architecture, the firmware shall be designed such that it works in 32-bit mode, and when the 32-bit operating system boots, it doesn't have to switch into protected mode because the CPU is already in protected mode. The same happens if the computer is 64-bit and you are loading a 64-bit operating system. This means that there is no 16-bit code in the OS source code anymore.

Of course this allows maximum utilization of the CPU during booting time. Further more, the OS can normally use UEFI routines in protected/long mode, which was impossible with the legacy BIOS. However, UEFI services are not designed for long-term usage; the OS should use UEFI services only on booting, and when the OS has done loading all device-drivers, it should tell the UEFI to give up control over the computer hardware to let the OS device drivers (which are usually more advanced than UEFI drivers) do their work.

2.3.3 A simple boot procedure

UEFI specification managed to make the boot procedure as simpler as possible. The following points illustrate how the boot procedure of UEFI is simpler and more powerful than legacy BIOS:

- In UEFI environment, no need to switch into the unreal mode anymore. The boot-loader can read the kernel from the disk and load it to memory above the famous 1MB boundary just using UEFI service routines.
- There is no need to activate A20 Line by the boot-loader; the firmware should do it for your OS.

- UEFI comes with a built-in INFLATE algorithm so that boot-loaders can decompress gz images without need to rewrite the algorithm again from scratch.
- A built-in FAT driver is provided by UEFI (by default). This allows the OS to put boot files (like kernel and boot-loader) onto a FAT disk drive. On booting, the boot-loader can load the kernel and device drivers from any FAT disk without need to have a FAT driver built inside the boot-loader.

As you can see, UEFI provides many facilities to make the design of the boot-loader easier and more reliable.

3 Code Examples

3.1 BIOS Code Example

Assuming we would like to write a boot-loader that does nothing except it prints "Hello World" on screen. In this example, assembly language is used. INT 0x10, like any service routine, needs some parameters to be passed through registers. To print some string, registers should load these values before invoking the interrupt:

- AH: The operation code of "Write string to console" (0x13).
- AL: Writing mode, which is 0x01 (a string is a stream of characters, attribute is stored in BL, and update cursor) in our case.
- BH: Should be loaded with page number (0 in our case).
- BL: The attribute (0x1F - Blue background, White foreground).
- CX: The count of the characters in the string (11 in our case).
- DH: Row position on screen (12 in our case).
- DL: Column position on screen (0 in our case).
- ES:BP: Pointer to string to write.

After loading the registers with appropriate values, the program shall execute (INT 0x10) instruction, which jumps to the portion of BIOS which is responsible for handling INT 0x10 requests (Video BIOS). After the VGA service routine is done, it returns back to the program, which halts (HLT) off the CPU [6].

```
org 7c00h

; initialize the stack
xor ax, ax
mov ss, ax
mov sp, ax

; set data and extra data segments
mov ds, ax
mov es, ax
```

```
; now load the parameters of print string routine
mov ah, 0x13 ; print string.
mov al, 0x01 ; update cursor and use BL as attribute.
mov bh, 0x00 ; page number.
mov bl, 0x1f ; attribute.
mov cx, 0x0b ; size of the string.
mov dh, 0x0C ; row.
mov dl, 0x00 ; column.
mov bp, lmsg ; ES:BP --> the string in memory.

; do it!
int 10h

; done
cli ; clear interrupts flag to ignore all IRQs.
hlt ; stop.

lmsg: db 'Hello World'

times 510-($-$$) db 0
dw 0AA55h
```

By assembling this program (using fasm) and installing the 512-byte binary file to the first sector of some floppy disk, you have a floppy disk which prints "Hello World" on row 12 of the screen on booting as shown in figure 5.

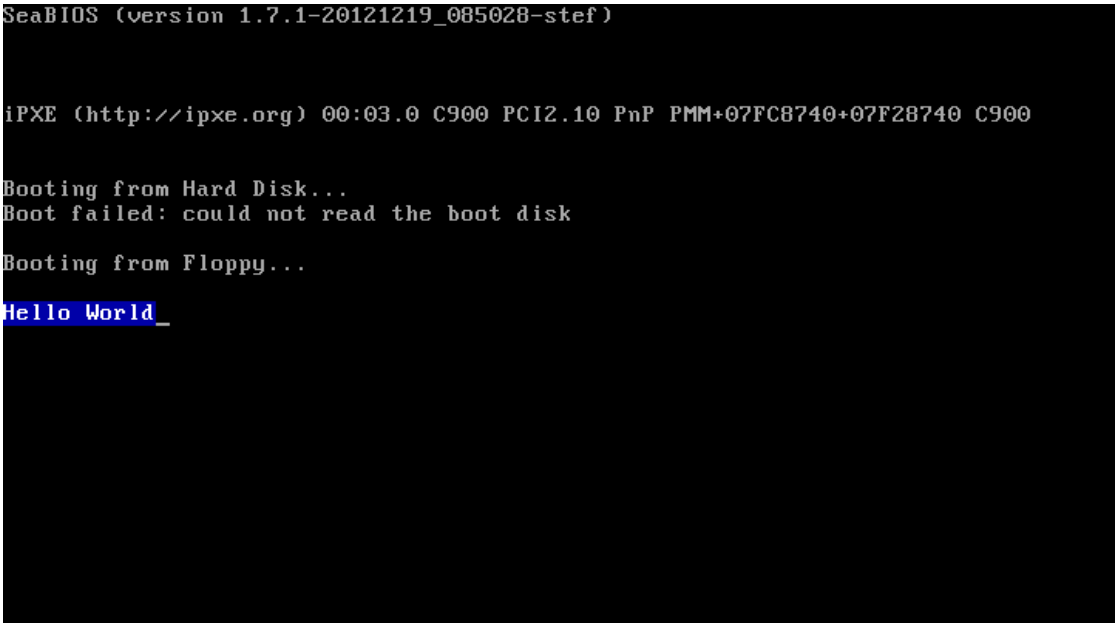

Figure 5: Hello world printed on row 12 of screen

This code example is shown here to explain how BIOS service routines are used by the operating system to access hardware without need for device drivers.

3.2 UEFI Code Example

In this sub-section, we are going to write a simple UEFI application that prints "Hello World" on screen. Writing applications for UEFI is different than writing them for legacy BIOS; software interrupts are not used in EFI. Rather than interrupts, the application, when starts up, will receive an "EFI system table" which contains pointers to various functions provided by the firmware.

The structure of EFI system table is specified by the UEFI standard specification. On Intel 64-bit architectures, the UEFI image receives a pointer to EFI system table within the RDX register. The table has an entry called "ConOut" which is actually a pointer to a structure of the type: "SIMPLE_TEXT_OUTPUT_INTERFACE".

The simple text output interface structure represents a specific console. In our case, the structure that is pointed by ConOut represents the ordinary console of the PC. In this structure, we can find the "OutputString" entry which is a pointer to OutputString() routine which we will call in order to print the "Hello World" string on the console.

The OutputString() routine takes 2 parameters:

1. RCX should contain a pointer to the SIMPLE_TEXT_OUTPUT_INTERFACE structure that represents the targeted console.
2. RDX should contain a pointer to some unicode string to be printed out.

So if we are coding in C and we want the program to call OutputString() routine with parameters ConOut and "Hello World", the code will look like this:

```
struct EFI_SYSTEM_TABLE *systable =
            (struct EFI_SYSTEM_TABLE *) RDX;
struct  SIMPLE_TEXT_OUTPUT_INTERFACE *console =
            systable->ConOut;
void () (*OutputString) = console->OutputString;
RCX = console;
RDX = L"Hello World";
OutputString();
return EFI_SUCCESS;
```

This piece of code can be translated into FASM as follows:

```
format pe64 dll efi
entry main
include 'efi.inc'

main:

        ; load the registers with appropriate values.
        mov rcx, [rdx + EFI_SYSTEM_TABLE.ConOut]
```

```
        mov rbx, [rcx + SIMPLE_TEXT_OUTPUT_INTERFACE.OutputString]
        lea rdx, [msg]

        ; do the call!
        call rbx

        ; return EFI_SUCCESS
        mov eax, EFI_SUCCESS
        retn

msg du "Hello world",0
```

The file (efi.inc) contains definitions for symbols that are used in the code above. As you can see, the code is very modular and easy to understand than the one written for BIOS. All other UEFI calls are done in a similar manner, without any need for the architecture-specific software interrupts and the complicated register assignments of BIOS calls.

This code can be compiled by FASM, after then, the object file (helloworld.efi) can be installed on a FAT disk drive and loaded by EFI shell. The screen-shot of figure 6 shows how to run the program from UEFI shell. Notice that OS loader is written in the same way, the difference is that it automatically loaded on start-up.

Figure 6: Running the hello world program from EFI shell.

Acknowledgements

We are very thankful to Dr. Mohamed Hussien for his continuous support and fruitful efforts. We also appreciate the help and support given by Eng. Mohamed Abd El-Aziz. We express our gratitude to Ahmed El-Khatib, who has helped us a lot with the English of this article.

References

[1] M. A. Mazidi and J. Mazidi, *80x86 IBM PC and Compatible Computers: Assembly Language, Design and Interfacing*. Prentice Hall PTR, 2000.

[2] *IBM PC/XT Technical Reference*. IBM, 1983.

[3] M. A. El-Aziz, "Quafios operating system." `http://www.quafios.com/`, 2013. [Online].

[4] *UEFI Standard Specification, Version 2.0*. Unified EFI, Inc., 2006.

[5] "Unified efi forum." `http://www.uefi.org/`. [Online].

[6] M. Perkel, "Ralf brown's interrupt list in html." `http://www.ctyme.com/rbrown.htm`. [Online].