Articles / Languages / ASM

VS2005 VS2008 C++ Windows Visual-Studio COM ASM

How to develop your own Boot Loader

Apriorit Inc

4.96/5 (281 votes)

30 May 2024 CPOL 16 min read 873.8K 20.5K

This article describes the first steps in low-level programming on the example of developing of simple boot loader

Download bootloader- 1.92 Mb

Contents

Who may be interested

Most of all I've written this article for those who have been always interested in the way the different things work. It is for those developers who usually create their applications in high-level languages such as C, C++ or Java, but faced with the necessity to develop something at low-level. We will consider low-level programming on the example of working at system loading.

We will describe what is going after you turn on a computer; how the system is loading. As the practical example we will consider how you can develop your own boot loader which is actually the first point of the system booting process.

What is Boot Loader

Boot loader is a program situated at the first sector of the hard drive; and it is the sector where the boot starts from. BIOS automatically reads all content of the first sector to the memory just after the power is turned on, and jump to it. The first sector is also called ***Master Boot Record.*** Actually it is not obligatory for the first sector of the hard drive to boot something. This name has been formed historically because developers used to boot their operating systems with such mechanism.

Be ready to go deeper

In this section I will tell about knowledge and tools you need to develop your own boot loader and also remind some useful information about system boot.

So what language you should know to develop Boot Loader

On the first stage on the computer work the control of hardware is performed mainly by means of BIOS functions known as interrupts. The implementation of interrupts is given only in Assembler – so it is great if you know it at least a little bit. But it’s not the necessary condition. Why? We will use the technology of “mixed code” where it is possible to combine high-level constructions with low-level commands. It makes our task a little simpler.

In this article the main development languages is C++. But if you have brilliant knowledge of C then it will be easy to learn required C++ elements. In general even the C knowledge will be enough but then you will have to modify the source code of the examples that I will descried here.

If you know Java or C# well unfortunately it won’t help for our task. The matter is that the code of Java and C# languages that is produced after compilation is intermediate. The special virtual machine is used to process it (Java Machine for Java, and .NET for C#) which transform intermediate code into processor instructions. After that transformation it can be executed. Such architecture makes it impossible to use mixed code technology – and we are going to use it to make our life easier, so Java and C# don’t work here.

So to develop the simple boot loader you need to know C or C++ and also it would be good if you know something about Assembler – language into which all high-level code is transformed it the end.

What compiler you need

To use mixed code technology, you need at least two compilers: one for Assembler and one for C/C++, as well as a linker to combine object files (.obj) into one executable file.

Now let's talk about the special aspects. There are two modes of processor operation: real mode and protected mode. The real mode is 16-bit and has some limitations. The protected mode is 32-bit and is fully used in the OS. At startup, the processor operates in 16-bit mode. So, to build a program and get an executable file, you will need a compiler and Assembler linker for 16-bit mode. For C/C++, you only need a compiler that can create object files for 16-bit mode.

Modern compilers are designed only for 32-bit programs, so we won't be able to use them.

I tried several free and commercial compilers for 16-bit mode and chose the Microsoft product. The compiler, along with the linker for Assembler, C, and C++, is included in the Microsoft Visual Studio 1.52 package, and it can also be downloaded from the company's official website. Some details about the compilers we need are provided below.

ML 6.15 – Microsoft assembler compiler for 16-bit mode;

LINK 5.16 – linker that can create .com files for 16-bit mode;

CL – C, C++ compiler for 16-bit mode.

You can also use several alternative options:

DMC – free compilation for Assembler, C, C++ for 16 and 32-bit modes from Digital Mars;

LINK – free linker for the DMC compiler;

There are also some products from Borland:

BCC 3.5 – C, C++ compiler that can create files for 16-bit mode;

TASM - assembler compiler for 16-bit mode;

TLINK – linker that can create .com files for 16-bit mode.

All code examples in this article were created using Microsoft tools

How the system boots

To solve our problem, we need to recall how the system boots.

Let's briefly consider how the system components interact during the boot process (see Fig. 1).

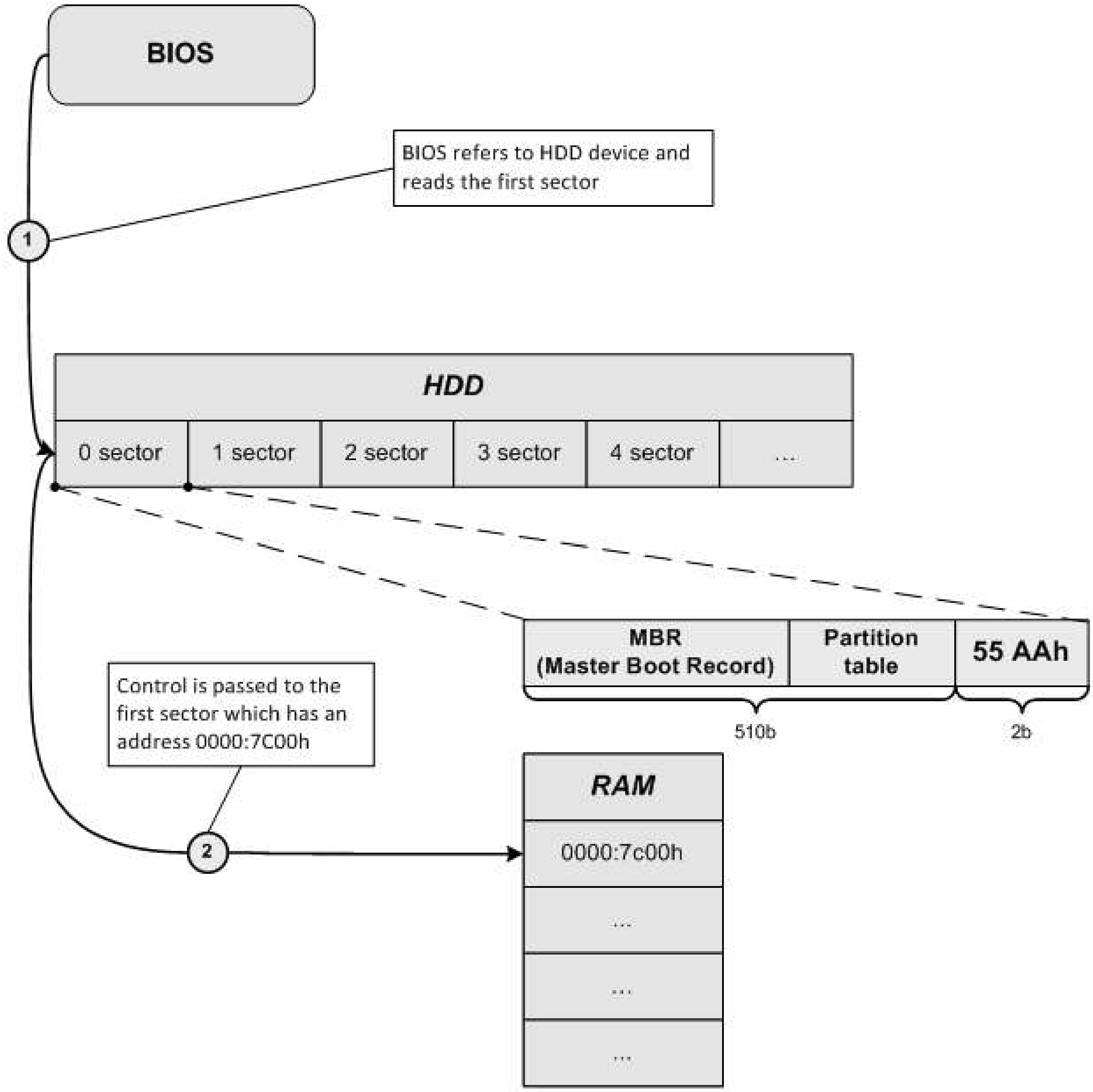

Fig. 1 - "How It Boots"

After control is transferred to address 0000:7C00, the Master Boot Record (MBR) starts its work and initiates the operating system boot process. You can learn more about the MBR structure, for example, here.

Let's code

In the following sections, we will directly engage in low-level programming by developing our own bootloader.

Program architecture

The bootloader we are developing is intended solely for learning purposes. Its tasks are as follows:

1. Proper loading into memory at address 0000:7C00.
2. Invocation of the BootMain function, developed in a high-level language.
3. Displaying the message "Hello, world..." on the display from a low level.

The program architecture is described in Fig. 2, followed by a textual description.

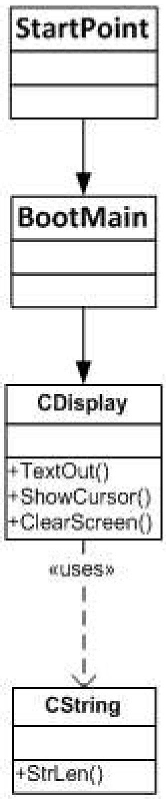

Fig. 2 - Program Architecture Description

The first entity is `StartPoint` that is developed purely in Assembler as far as high-level languages don't have the necessary instructions. It tells compiler what memory model should be used, and what address the loading to the RAM should be performed by after the reading from the disk. It also corrects processor registers and passes control to the `BootMain` that is written in high-level language.

Next entity– `BootMain` – is an analogue of `main` that is in its turn the main function where all program functioning is concentrated.

`CDisplay` and `CString` classes take care of functional part of the program and show message on the screen. As you can see from the Fig.2 `CDisplay` class uses `CString`class in its work.

Development environment

Here I use the standard development environment ***Microsoft Visual Studio 2005*** or ***2008***. You can use any other tools but I made sure that these two with some settings made the compiling and work easy and handy.

First we should create the project of ***Makefile Project*** type where the main work will be performed (see Fig.3).

File->New\Project->General\Makefile Project

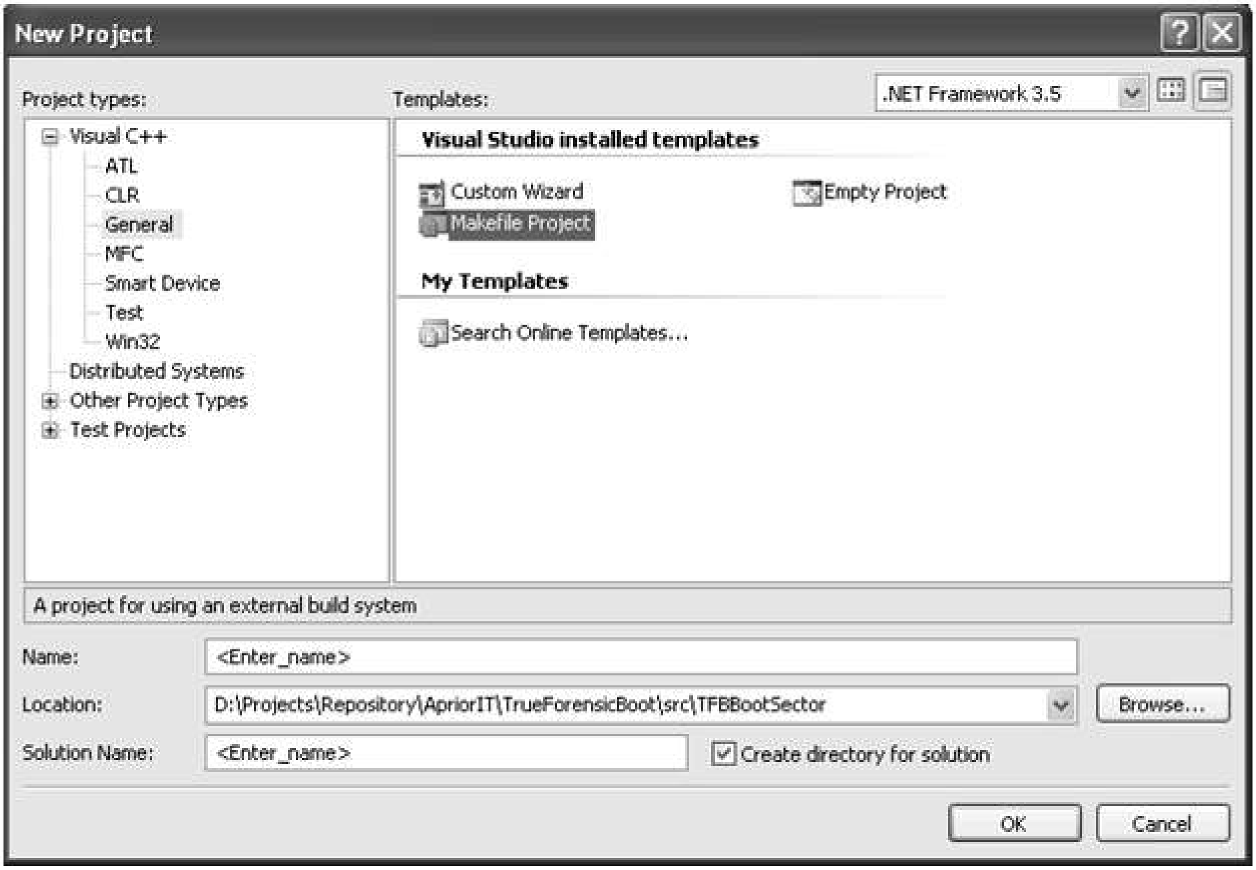

Fig.3 – Create the project of Makefile type

BIOS interrupts and screen clearing

To show our message on the screen we should clear it first. We will use special BIOS interrupt for this purpose.

BIOS proposes a number of interrupts for the work with computer hardware such as video adapter, keyboard, disk system. Each interrupt has the following structure:

ASM

```
int [number_of_interrupt];
```

where number_of_interrupt is the number of interrupt

Each interrupt has the certain number of parameters that should be set before calling it. The `ah` processor register is always responsible for the number of function for the current interrupt, and the other registers are usually used for the other parameters of the current operation. Let's see how the work of `int 10h` interrupt is performed in Assembler. We will use the 00 function that changes the video mode and clears screen:

ASM

```
mov al, 02h ; setting  the graphical mode 80x25(text)
mov ah, 00h ; code  of function of changing video mode
int 10h   ; call  interruption
```

We will consider only those interrupts and functions that will be used in our application. We will need:

ASM

```
int 10h, function 00h – performs changing of video mode and clears  screen;
int 10h, function 01h – sets the cursor type;
int 10h, function 13h – shows the string on the screen;
```

«Mixed code»

Compiler for C++ supports the inbuilt Assembler i.e. when writing code in igh-level language you can use also low level language. Assembler Instructions that are used in the high level code are also called ***asm insertions***. They consist of the key word __`asm` and the block of the Assembler instructions in braces:

C++

```
__asm ;  key word that shows the beginning of the asm insertion
  { ;  block beginning

  … ; some asm code
  } ;  end of the block
```

To demonstrate mixed code let's use the previously mentioned Assembler code that performed the screen clearing and combine it with C++ code.

C++

```
void ClearScreen()
{
 __asm
{
 mov al, 02h ; setting the graphical mode 80x25(text)
mov ah, 00h ; code  of function of changing video mode
int 10h   ; call interrupt
}
}
```

`CString` implementation

`CString` class is designed to work with strings. It includes `Strlen()` method that obtains pointer to the string as the parameter and returns the number of symbols in that string.

C++ Shrink ▲

```
// CString.h

#ifndef __CSTRING__
#define __CSTRING__

#include "Types.h"

class CString
{
public:
    static byte Strlen(
        const char far* inStrSource
        );
};

#endif // __CSTRING__

// CString.cpp

#include "CString.h"

byte CString::Strlen(
        const char far* inStrSource
        )
{
        byte lenghtOfString = 0;

        while(*inStrSource++ != '\0')
        {
            ++lenghtOfString;
        }
        return lenghtOfString;
}
```

`CDisplay` implementation

`CDisplay` class is designed for the work with the screen. It includes several methods:

1) `TextOut()` – it prints the string on the screen.
2) `ShowCursor()` – it manages the cursor representation on the screen: show, hide.
3) `ClearScreen()` – it changes the video mode and thus clears screen.

C++ Shrink ▲

```
  // CDisplay.h

#ifndef __CDISPLAY__
#define __CDISPLAY__
```

```
//
// colors for TextOut func
//

#define BLACK           0x0
#define BLUE            0x1
#define GREEN           0x2
#define CYAN            0x3
#define RED             0x4
#define MAGENTA         0x5
#define BROWN           0x6
#define GREY            0x7
#define DARK_GREY           0x8
#define LIGHT_BLUE      0x9
#define LIGHT_GREEN     0xA
#define LIGHT_CYAN      0xB
#define LIGHT_RED             0xC
#define LIGHT_MAGENTA       0xD
#define LIGHT_BROWN     0xE
#define WHITE           0xF

#include "Types.h"
#include "CString.h"

class CDisplay
{
public:
    static void ClearScreen();

    static void TextOut(
        const char far* inStrSource,
        byte            inX = 0,
        byte            inY = 0,
        byte            inBackgroundColor   = BLACK,
        byte            inTextColor         = WHITE,
        bool            inUpdateCursor      = false
        );

    static void ShowCursor(
        bool inMode
        );
};

#endif // __CDISPLAY__

// CDisplay.cpp

#include "CDisplay.h"

void CDisplay::TextOut(
        const char far* inStrSource,
        byte            inX,
        byte            inY,
        byte            inBackgroundColor,
        byte            inTextColor,
        bool            inUpdateCursor
        )
```

```
{
    byte textAttribute = ((inTextColor) | (inBackgroundColor << 4));
    byte lengthOfString = CString::Strlen(inStrSource);

    __asm
    {
        push    bp
        mov     al, inUpdateCursor
        xor     bh, bh
        mov     bl, textAttribute
        xor     cx, cx
        mov     cl, lengthOfString
        mov     dh, inY
        mov     dl, inX
        mov     es, word ptr[inStrSource + 2]
        mov     bp, word ptr[inStrSource]
        mov     ah, 13h
        int     10h
        pop     bp
    }
}
void CDisplay::ClearScreen()
{
    __asm
    {
        mov     al, 02h
        mov     ah, 00h
        int     10h
    }
}

void CDisplay::ShowCursor(
        bool inMode
        )

{
    byte flag = inMode ? 0 : 0x32;

    __asm
    {
        mov     ch, flag
        mov     cl, 0Ah
        mov     ah, 01h
        int     10h
    }
}
```

`Types.h` implementation

`Types.h` is the header file that includes definitions of the data types and macros.

C++

```
// Types.h
```

```
#ifndef __TYPES__
#define __TYPES__

typedef unsigned char   byte;
typedef unsigned short  word;
typedef unsigned long   dword;
typedef char            bool;

#define true            0x1
#define false           0x0

#endif // __TYPES__
```

`BootMain.cpp` implementation

`BootMain()` is the main function of the program that is the first entry point (analogue of `main()`). Main work is performed here.

C++

```
// BootMain.cpp

#include "CDisplay.h"

#define HELLO_STR               "\"Hello, world…\", from low-level..."

extern "C" void BootMain()
{
    CDisplay::ClearScreen();
    CDisplay::ShowCursor(false);

    CDisplay::TextOut(
        HELLO_STR,
        0,
        0,
        BLACK,
        WHITE,
        false
        );

    return;
}
```

`StartPoint.asm` implementation

ASM

Shrink ▲

```
;------------------------------------------------------------
.286                               ; CPU type
;------------------------------------------------------------
.model TINY                        ; memory of model
;---------------------- EXTERNS -----------------------------
extrn               _BootMain:near     ; prototype of C func
;------------------------------------------------------------
```

```
;-------------------------------------------------------------
.code
org             07c00h          ; for BootSector
main:
                jmp short start    ; go to main
                nop

;----------------------- CODE SEGMENT -----------------------
start:
        cli
        mov ax,cs               ; Setup segment registers
        mov ds,ax               ; Make DS correct
        mov es,ax               ; Make ES correct
        mov ss,ax               ; Make SS correct
        mov bp,7c00h
        mov sp,7c00h            ; Setup a stack
        sti
                                ; start the program
        call           _BootMain
        ret

        END main                ; End of program
```

Let's assemble everything

Creation of COM file

Now when the code is developed we need to transform it to the file for the 16-bit OS. Such files are ***.com files***. We can start each of compilers (for Assembler and C, C++) from the command line, transmit necessary parameters to them and obtain several object files as the result. Next we start linker to transform all .obj files to the one executable file with .com extension. It is working way but it's not very easy.

Let's automate the process. In order to do it we create .bat file and put commands with necessary parameters there. Fig.4 represents the full process of application assembling.

Fig.4 – Process of program compilation

Build.bat

Let’s put compilers and linker to the project directory. In the same directory we create .bat file and fill it accordingly to the example (you can use any directory name instead of VC152 where compilers and linker are situated):

.\VC152\CL.EXE /AT /G2 /Gs /Gx /c /Zl *.cpp

.\VC152\ML.EXE /AT /c *.asm

.\VC152\LINK.EXE /T /NOD StartPoint.obj bootmain.obj cdisplay.obj cstring.obj

del *.obj

Assembly automation

As the final stage in this section we will describe the way how to turn Microsoft Visual Studio 2005, 2008 into the development environment with any compiler support. Go to the Project Properties: ***Project->Properties->Configuration Properties\General->Configuration Type***.

Configuration Properties tab includes three items: ***General***, ***Debugging***, ***NMake***. Go to ***NMake*** and set the path to the ***build.bat*** in the ***Build Command Line*** and ***Rebuild Command Line*** fields – Fig.5.

Fig.5 –NMake project settings

If everything is correct then you can compile in the common way pressing ***F7*** or ***Ctrl + F7***. At that all attendant information will be shown in the Output window. The main advantage here is not only the assembly automation but also navigation thru the code errors if they happen.

Testing and Demonstration

This section will tell how to see the created boot loader in action, perform testing and debug.

How to test boot loader

You can test boot loader on the real hardware or using specially designed for such purposes virtual machine – VmWare. Testing on the real hardware gives you more confidence that it works while testing on the virtual machine makes you confident that it just can work. Surely we can say that VmWare is great method for testing and debug. We will consider both methods.

First of all we need a tool to write our boot loader to the virtual or physical disk. As far as I know there a number of free and commercial, console and GUI applications. I used ***Disk Explorer for NTFS 3.66*** (version for FAT that is named Disk Explorer for FAT) for work in Windows and ***Norton Disk Editor 2002*** for work in MS-DOS.

I will describe only Disk Explorer for NTFS 3.66 because it is the simplest method and suits our purposes the most.

Testing with the virtual machine VmWare

Creation of the virtual machine

We will need VmWare program version 5.0, 6.0 or higher. To test boot loader we will create the new virtual machine with minimal disk size for example 1 Gb. We format it for NTFS file system. Now we need to map the formatted hard drive to VmWare as the virtual drive. To do it:

File->Map or Disconnect Virtual Disks...

After that the window appears. There you should click Map button. In the next appeared window you should set the path to the disk. Now you can also chose the letter for the disk- see Fig.6.

Fig.6 – Parameters of virtual disk mapping

Don’t forget to uncheck the ***“Open file in read-only mode (recommended)”*** checkbox. When checked it indicates that the disk should be opened in read-only mode and prevent all recording attempts to avoid data corruption.

After that we can work with the disk of virtual machine as with the usual Windows logical disk. Now we should use Disk Explorer for NTFS 3.66 and record boot loader by the physical offset 0.

Working with Disk Explorer for NTFS

After program starts we go to our disk (***File->Drive***). In the window appeared we go to the ***Logical Drives*** section and chose disk with the specified letter (in my case it is Z) – see Fig.7.

Fig.7 – choosing disk in Disk Explorer for NTFS

Now we use menu item ***View*** and ***As Hex*** command. It the appeared window we can see the information on the disk represented in the 16-bit view, divided by sectors and offsets. There are only 0s as soon as the disk is empty at the moment. You can see the first sector on the Fig.8.

Fig.8 – Sector 1 of the disk

Now we should write our boot loader program to this first sector. We set the marker to position 00 as it is shown on the Fig.8. To copy boot loader we use ***Edit*** menu item, ***Paste from file*** command. In the opened window we specify the path to the file and click ***Open***. After that the content of the first sector should change and look like it's shown on the Fig.9 – if you haven't changed anything in the example code, of course.

You should also write signature 55AAh by the 1FE offset from the sector beginning. If you don't do it BIOS will check the last two bytes, won't find the mentioned signature and will consider this sector as not the boot one and won't read it to the memory.

To switch to the edit mode press ***F2*** and write the necessary numbers –55AAh signature. To leave edit mode press ***Esc***.

Now we need to confirm data writing.

Fig.9 – Boot Sector appearance

To apply writing we go to ***Tools->Options.*** Window will appear; we go to the ***Mode*** item and chose the method of writing - ***Virtual Write*** and click ***Write*** button – Fig.10.

Fig.10 – Choosing writing method in Disk Explorer for NTFS

A great number of routine actions are finished at last and now you can see what we have been developing from the very beginning of this article. Let's return to the VwWare to disconnect the virtual disk (***File->Map or Disconnect Virtual Disks***... and click ***Disconnect***).

Let's execute the virtual machine. We can see now how from the some depth, from the kingdom of machine codes and electrics the familiar string appears ***""Hello, world...", from low-level..."*** – see Fig.11.

Fig.11 – “Hello world…”

Testing on the real hardware

Testing on the real hardware is almost the same as on the virtual machine except the fact that if something doesn’t work you will need much more time to repair it than to create the new virtual machine. To test boot loader without the threat of existent data corruption (everything can happen), I propose to use flash drive, but first you should reboot your PC, enter BIOS and check if it supports boot from the flash drive. If it does than everything is ok. If it does not than you have to limit your testing to virtual machine test only.

The writing of boot loader to the flash disk in Disk Explorer for NTFS 3.66 is the same to the process for virtual machine. You just should choose the hard drive itself instead of its logical section to perform writing by the correct offset – see Fig.12.

Fig.12 – Choosing physical disk as the device

Debug

If something went wrong – and it usually happens – you need some tools to debug your boot loader. I should say at once that it is very complicated, tiring and time-eating process. You will have to grasp in the Assembler machine codes – so good knowledge of this language is required. Any way I give a list of tools for this purpose:

TD (Turbo Debugger) – great debugger for 16-bit real mode by Borland.

CodeView – good debugger for 16-bit mode by Microsoft.

D86 – good debugger for 16-bit real mode developed by Eric Isaacson – honored veteran of development for Intel processor in Assembler.

Bocsh – program-emulator of virtual machine that includes debugger of machine commands.

Information Sources

"Assembly **Language for Intel-Based Computers"** ***by Kip R. Irvine*** is the great book that gives good knowledge of inner structure of the computer and development in Assembler. You ca also find information about installation, configuration and work with the MASM 6.15 compiler.

This link will guide you to the BIOS interrupt list: http://en.wikipedia.org/wiki/BIOS_interrupt_call

Conclusion

In this article, we looked at what a bootloader is, how the BIOS works, and how system components interact when booting the system. The practical part provided information on how to develop your own simple bootloader. We demonstrated the mixed code technology and the build automation process using Microsoft Visual Studio 2005, 2008.

Of course, this is a small piece compared to the huge topic of low-level programming, but if this article interested you, then great.

License