The booting process

by Gergor Brunmar

Things that are good to know...

So, you want to make your own OS, but has no clue where to start?! First, you need to find the right tools. I use mixed assembly and C-code to make MuOS. The tools I use are:

- For assembly: Nasm
- For C-code: Gcc (I use Djgpp, but that's because I do my work on the Win32-platform, although, everything *should* work on any platform)
- For PC-emulation: Bochs (You don't want to reboot every time you want to test something)

Now, on to the good stuff =). The aim for our OS is to be 32-bit. For those who now thinks: '*Hey?! Isn't every 386+, 32-bits processors? Why aim lower that a 386?*': - There's a reason!

Back when the 386 was introduced, the 32-bits were a brand new feature. For all thoses people who spent many $$$ for their 16-bit programs, backwards compability was a must. Intel decided to make the 386 start in 16-bit mode and if the Operating System supported 32-bits, it would have to change from 16-bit mode to 32-bit mode manually. And everybody lived happily ever after... Okay, present day. Computers still boot in 16-bit compability mode. There are hardly any 16-bit operating systems left (DOS is 16-bit, and there are still users, so I can't say that there are none).

There are a few things that change dramatically when 32-bit mode are activated and it's just because of these reasons that we want to have a 32-bit OS (some of the features can be accessed in 16-bit mode, but it requires some special techniques to be used):

- **Access to 4 GB of memory** - At startup, the processor is in a so called *Real mode* (16-bit). This limits the memory access to about 1 MB. Howevery, in *Protected mode* (32-bit), memory up to 4 GB(!) can be accessed. That's a lot more than 1 MB =).
- **Memory protection**
  - The Protected mode makes reason for its name in memory protection. Memory can be write-protected so that critical sections cannot be touched (OS parts or other application's data for example).
- **Multitasking** - The processor has built-in support for task switching. Multitasking is not accually parallell processes, but scheduled processing time. This feature saves the state of the registers and loads it with the next task's register values. (This can be done with software, but it's faster in hardware most of the time)

There are more features, but these are the most important ones (if you ask me). That should cover the basics you'll need to know. On to the good stuff - **Writing a *boot sector*!**

The fun and dangerous part!

Create a plain text file called 'booting.asm'. The first thing we should do, is to tell the compiler, we're compiling to 16-bit instructions. Remember, at the start-up, the computer operates in Real mode (16-bit).

```
[BITS 16]
```

Then we tell the compiler, where in the memory our program is resident. In Nasm, this is done by the ORG command. '*Why does the compiler need to know that?*', you ask. Take a look at this piece of code:

```
[ORG 0x7C00]
   mov ax, [label]
label:
   dw 0
```

The *mov* instruction is assembled into '*mov ax, 0x7C03*', instead of '*mov ax, 0x0003*'. This had not been necessary if our program was a '*normal*' application, but now, we're making a *boot sector*. The number I picked, 0x7C00, is the memory address the BIOS puts the boot sector it finds in, so this is where our programs is resident. This adds the line:

```
[ORG 0x7C00]
```

Now we want to write a message on the screen. To make it simple, we use the BIOS interrupts available in Real mode. Int 10h has all the functionallity we need. If we put 0Eh into the AH register, we tell the BIOS that we want to put a single character onto the screen. The BIOS then takes the ASCII value of AL, combineds this with the color information in BH and prints the character. The BL register is used to set a page number, but we're not using this, so just set it to 0 and ignore it. Now our code for this:

```
mov ah, 0Eh
mov al, 'A'
mov bh, 0Fh
mov bl, 0
int 10h
```

The BH register holds, as said before, the color attribute for the character. I didn't explain why I put 0Fh in BH, but it holds the color code for the character. This seems not to be supported by all BIOSes, but give it a try if you want to (doesn't work in Bochs). The color codes are as follows:

Value	**Color**	**Value**	**Color**
00h	Black	*08h*	Dark gray
01h	Blue	*09h*	Bright blue
02h	Green	*0Ah*	Bright green
03h	Cyan	*0Bh*	Bright cyan
04h	Red	*0Ch*	Pink
05h	Magenta	*0Dh*	Bright magenta
06h	Brown	*0Eh*	Yellow
07h	Gray	*0Fh*	White

To get more information about BIOS interrupts, there's a complete listing made by Ralf Brown. You'll find his interrupt list here.

After we displayed our 'A' we just hang... This is done by making a jump to a jump, to a jump, to a jump, etc...:

```
hang:
jmp hang
```

To get the BIOS to recognize the file as a valid boot sector, the word at address 0x510 must be set to 55AAh. First we fill up the rest of the file with zeros and then we add our word. This is done by adding the lines:

```
times 510-($-$$) db 0
dw 55AAh
```

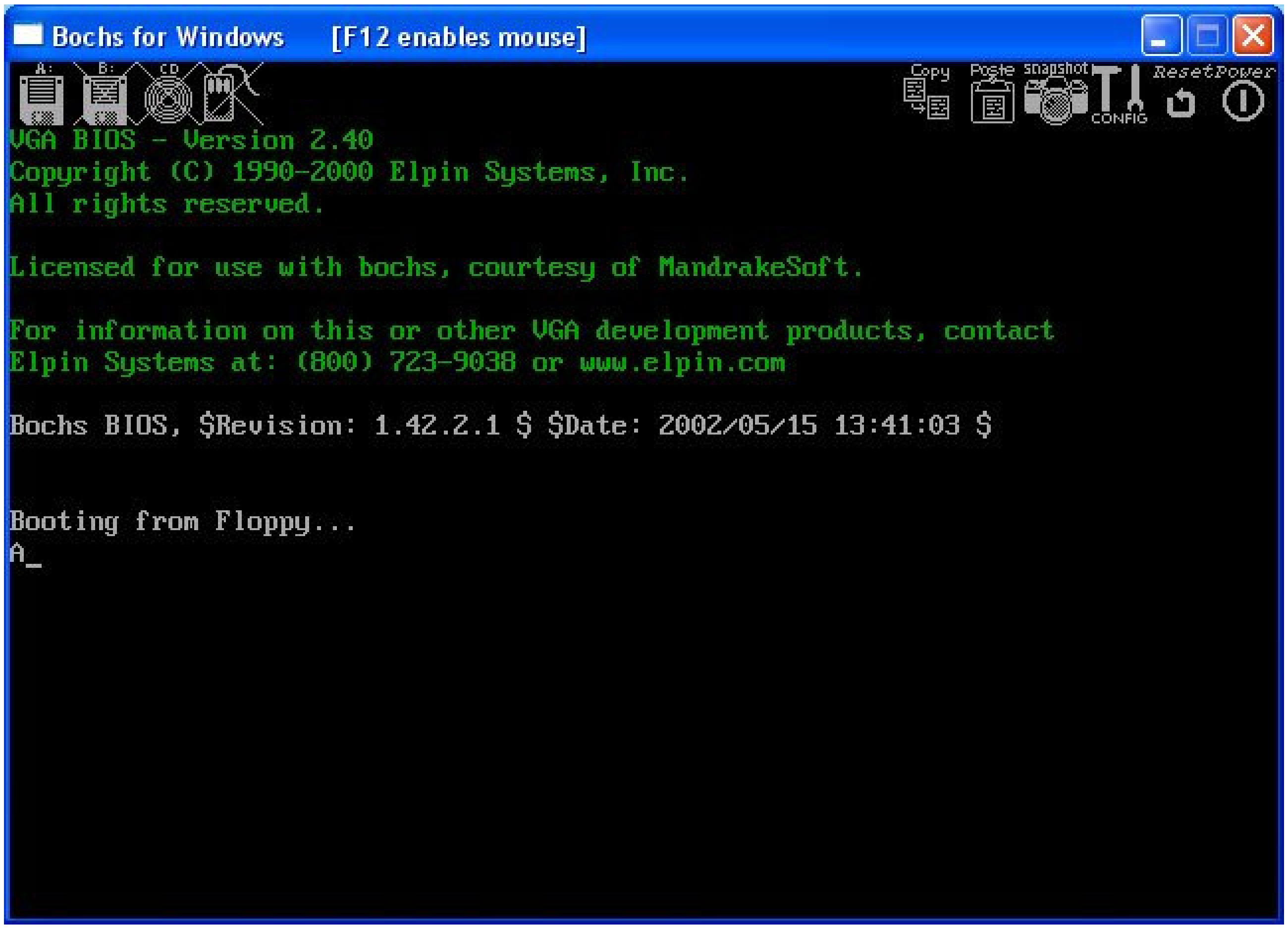
Putting it together

Now we have a valid boot sector, but we must only compile it first. To compile it to a plain binary file, which the BIOS can read, we use nasm (or nasmw on windows) with the switches:

```
nasm -f bin booting.asm -o booting.bin
```

The '*-f bin*' specifies the format to plain binary. You can choose a different output name, but I chose '*booting.bin*', because it was a logical name =). Now we are all set to test it. Copy the file into Bochs' directory and run it with the booting.bin as a floppy and we're done!

Here's a screenshot from Bochs, running the tutorial's code:



```
VGA BIOS - Version 2.40
Copyright (C) 1990-2000 Elpin Systems, Inc.
All rights reserved.

Licensed for use with bochs, courtesy of MandrakeSoft.

For information on this or other VGA development products, contact
Elpin Systems at: (800) 723-9038 or www.elpin.com

Bochs BIOS, $Revision: 1.42.2.1 $ $Date: 2002/05/15 13:41:03 $


Booting from Floppy...
A_
```

Download the complete source for this tutorial.
Download my example configuration file for Bochs (paths to the BIOS may have to be changed, if you're using another distributions than Win32 1.4.1).

Any comments, improvments or found errors? Mail me: gregor.brunmar@home.se.

Goto the next tutorial.