
**"Linux Gazette...*making Linux just a little more fun!*"**

---

# Writing Your Own Toy OS (Part I)

**By Krishnakumar R.**

---

*This article is a hands-on tutorial for building a small boot sector. The first section provides the theory behind what happens at the time the computer is switched on. It also explains our plan. The second section tells all the things you should have on hand before proceeding further, and the third section deals with the programs. Our little startup program won't actually boot Linux, but it will display something on the screen.*

---

## 1. Background

## 1.1 The Fancy Dress

The microprocessor controls the computer. At startup, every microprocessor is just another 8086. Even though you may have a brand new Pentium, it will only have the capabilities of an 8086. From this point, we can use some software and switch processor to the infamous *protected mode* . Only then can we utilize the processor's full power.

## 1.2 Our Role

Initially, control is in the hands of the *BIOS*. This is nothing but a collection of programs that are stored in ROM. BIOS performs the *POST* (Power On Self Test). This checks the integrity of the computer (whether the peripherals are working properly, whether the keyboard is connected, etc.). This is when you hear those beeps from the computer. If everything is okay, BIOS selects a boot device. It copies the first sector (boot sector) from the device, to address location *0x7C00*. The control is then transferred to this location. The boot device may be a floppy disk, CD-ROM, hard disk or some device of your choice. Here we will take the boot device to be a floppy disk. If we had written some code into the boot sector of the floppy, our code would be executed now. Our role is clear: just write some programs to the boot sector of the floppy.

## 1.3 The Plan

First write a small program in 8086 assembly (don't be frightened; I will teach you how to write it), and copy it to the boot sector of the floppy. To copy, we will code a C program. Boot the computer with that floppy, and then enjoy.

# 2. Things You Should Have

**as86**

This is an assembler. The assembly code we write is converted to an object file by this tool.

**ld86**

This is the linker. The object code generated by as86 is converted to actual machine language code by this tool. Machine language will be in a form that 8086 understands.

**gcc**

The C compiler. For now we need to write a C program to transfer our OS to the floppy.

**A free floppy**

A floppy will be used to store our operating system. This also is our boot device.

**Good Old Linux box**

You know what this is for.

as86 and ld86 will be in most of the standard distributions. If not, you can always get them from the site http://www.cix.co.uk/~mayday/. Both of them are included in single package, bin86. Good documentation is available at www.linux.org/docs/ldp/howto/Assembly-HOWTO/as86.html.

# 3. 1, 2, 3, Start!

## 3.1 The Boot Sector

Grab your favourite editor and type in these few lines.

```
entry start
start:
      mov ax,#0xb800
      mov es,ax
      seg es
      mov [0],#0x41
      seg es
      mov [1],#0x1f
loop1: jmp loop1
```

This is an assembly language that as86 will understand. The first statement specifies the entry point where the control should enter the program. We are stating that control should initially go to label *start*. The 2nd line depicts the location of the label *start* (don't forget to put ":" after the start). The first statement that will be executed in this program is the statement just after *start*.

0xb800 is the address of the video memory. The # is for representing an immediate value. After the execution of

```
mov ax,#0xb800
```

register ax will contain the value 0xb800, that is, the address of the video memory. Now we move this value to the *es* register. *es* stands for the extra segment register. Remember that 8086 has a segmented architecture. It has segments like code segments, data segments, extra segments, etc.--hence the segment registers cs, ds, es. Actually, we have made the video memory our extra segment, so anything written to extra segment would go to video memory.

To display any character on the screen, you need to write two bytes to the video memory. The first is the ascii value you are going to display. The second is the attribute of the character. Attribute has to do with which colour should be used as the foreground, which for the background, should the char blink and so on. *seg es* is actually a prefix that tells which instruction is to be executed next with reference to *es* segment. So, we move value 0x41, which is the ascii value of character A, into the first byte of the video memory. Next we need to move the attribute of the character to the next byte. Here we enter 0x1f, which is the value for representing a white character on a blue background. So if we execute this program, we get a white A on a blue background. Finally, there is the loop. We need to stop the execution after the display of the character, or we have a loop that loops forever. Save the file as *boot.s*.

The idea of video memory may not be very clear, so let me explain further. Suppose we assume the screen consists of 80 columns and 25 rows. So for each line we need 160 bytes, one for each character and one for each character's attribute. If we need to write some character to column 3 then we need to skip bytes 0 and 1 as they is for the 1st column; 2 and 3 as they are for the 2nd column; and then write our ascii value to the 4th byte and its attribute to the 5th location in the video memory.

# 3.2 Writing Boot Sector to Floppy

We have to write a C program that copies our code (OS code) to first sector of the floppy disk. Here it is:

```
#include <sys/types.h> /* unistd.h needs this */
#include <unistd.h>    /* contains read/write */
#include <fcntl.h>

int main()
{
        char boot_buf[512];
        int floppy_desc, file_desc;


        file_desc = open("./boot", O_RDONLY);
        read(file_desc, boot_buf, 510);
        close(file_desc);
```

```
        boot_buf[510] = 0x55;
        boot_buf[511] = 0xaa;

        floppy_desc = open("/dev/fd0", O_RDWR);
        lseek(floppy_desc, 0, SEEK_CUR);
        write(floppy_desc, boot_buf, 512);
        close(floppy_desc);
}
```

First, we open the file *boot* in read-only mode, and copy the file descripter of the opened file to variable *file_desc*. Read from the file 510 characters or until the file ends. Here the code is small, so the latter case occurs. Be decent; close the file.

The last four lines of code open the floppy disk device (which mostly would be /dev/fd0). It brings the head to the beginning of the file using *lseek*, then writes the 512 bytes from the buffer to floppy.

The man pages of read, write, open and lseek (refer to man 2) would give you enough information on what the other parameters of those functions are and how to use them. There are two lines in between, which may be slightly mysterious. The lines:

```
boot_buf[510] = 0x55;
boot_buf[511] = 0xaa;
```

This information is for BIOS. If BIOS is to recognize a device as a bootable device, then the device should have the values 0x55 and 0xaa at the 510th and 511th location. Now we are done. The program reads the file *boot* to a buffer named boot_buf. It makes the required changes to 510th and 511th bytes and then writes boot_buf to floppy disk. If we execute the code, the first 512 bytes of the floppy disk will contain our boot code. Save the file as *write.c*.

## 3.3 Let's Do It All

To make executables out of this file you need to type the following at the Linux bash prompt.

```
as86 boot.s -o boot.o

ld86 -d boot.o -o boot

cc write.c -o write
```

First, we assemble the *boot.s* to form an object file *boot.o*. Then we link this file to get the final file *boot*. The *-d* for ld86 is for removing all headers and producing pure binary. Reading man pages for as86 and ld86 will clear any doubts. We then compile the C program to form an executable named *write*.

Insert a blank floppy into the floppy drive and type

```
./write
```

Reset the machine. Enter the BIOS setup and make floppy the first boot device. Put the floppy in the drive and watch the computer boot from your boot floppy.

Then you will see an 'A' (with white foreground color on a blue background). That means that the system has booted from the boot floppy we have made and then executed the boot sector program we wrote. It is now in the infinite loop we had written at the end of our boot sector. We must now reboot the computer and remove the our boot floppy to boot into Linux.

From here, we'll want to insert more code into our boot sector program, to make it do more complex things (like using BIOS interrupts, protected-mode switching, etc). The later parts (PART II, PART III etc. ) of this article will guide you on further improvements. Till then GOOD BYE !

**Krishnakumar R.**

*Krishnakumar is a final year B.Tech student at Govt. Engg. College Thrissur, Kerala, India. His journey into the land of Operating systems started with module programming in linux . He has built a routing operating system by name GROS.(Details available at his home page: www.askus.way.to ) His other interests include Networking, Device drivers, Compiler Porting and Embedded systems.*

---


**Published in Issue 77 of *Linux Gazette*, April 2002**

---

TABLE OF CONTENTS | FRONT PAGE | TALKBACK | FAQ