# Rolling Your Own Bootloader

From OSDev Wiki

**Difficulty level**

◻◻◻◻
Not rated

Some people prefer to use their own software for everything. This page attempts to describe what steps to take when you write your own bootloader. Before you start writing one, it is best that you know the background theory.

## Contents

## What and Why

### Disclaimer

Okay. We have GRUB, we have Bare bones and C++ Bare Bones but still people complain we don't have a page explaining how the bootloader can be coded.

I won't try to give you full code that works because if that was what you were looking for, you'd be using one of the premade bootloaders instead. This page plans to tell you what is needed and what could be wished in a bootloader, and optionally points at parts of the FAQ that can help you achieving the goals.

Whether or not you'll use your own bootloader or reuse an existing tool is completely up to you. If you get the feeling you don't understand a thing, make sure you read our page about the Boot Sequence first ;)
A good reason to have a custom bootloader would be a custom filesystem.

### What you need to do

The bootloader ultimately has to bring the kernel (and all the kernel needs to bootstrap) in memory, switch to an environment that the kernel will like and then transfer control to the kernel.

As the scope of this article is protected mode C kernels, I'll assume that "an environment the kernel will like" means Protected Mode, with kernel and additional components being stored at their 'favorite', compile-time known locations, with a wide-enough stack ready and BSS section cleared.

### What you could wish to add

Since the bootloader runs in Real Mode, it has easier access to BIOS resources and functions. Therefore it's a good place to perform memory map detection, detecting available video modes, loading additional files etc. The bootloader will collect this information and present it in a way the kernel will be able to understand

## Loading ... Please wait ...

### Where will you load your kernel ?

You will have to decide where in memory you are going to load your kernel. Your kernel generally depends on it.

In Real Mode, the easiest is to stay below the 1MB barrier, which means you practically have 512KB of memory to load things. You may wish the kernel to be loaded at a well-known position, say 0x10000 physical (es=0x1000, bx=0 when calling INT13h).

If your kernel is bigger (or is expected to grow bigger) than this, you'll probably prefer to have the kernel above the 1MB barrier, which means you need to activate A20 gate and switch to Unreal Mode to load the kernel (with A20 alone, you cannot have more than 64K above 1MB).

Note that BIOS will still be unable to write to memory above 1MB, so you need to read stuff in a buffer below 1MB and then perform a rep movsd to place the data where they ultimately should go.

### How will you find your kernel ?

The bits of your kernel are somewhere on some disk (presumably the booting disk, but this is not mandatory). Question is: where on the disk ? Is it a regular file on a FAT-formatted floppy ? is it a collection of consecutive sectors in the "reserved area" of the FAT12 floppy (in which case you may need a dedicated tool to format the disk and install the kernel on it) ? Or is the floppy simply left unformatted and kernel pasted directly with a disk image tool.

All the above options are possible. Maybe the one I'd choose myself would be to reserve enough space on a FAT12 floppy to store the *list of sectors* used by the kernel file. The "advantage" of being fully-FAT12 is that you don't need to re-write the bootsector every time you rewrite the kernel.

### What else could you need to load ?

That mainly depends on what's in your kernel. Linux, for instance, requires an additional 'initrd' file that will contain the 'initialization process' (as user level). If your kernel is modular and if File Systems are understood by some modules, you need to load the modules along with the kernel. Same goes for 'microkernel services' like disk/files/memory services, etc.

### What if I get beyond the 512 bytes of the boot sector ?

Use GRUB ;) -- nah, kidding ... just make sure the first 512 bytes are able to load the rest of your loader and you're safe. Some do this with a separate "second stage" loader, others by really inserting a '512-bytes' break in their ASM code, making sure the rest of the loader is put after the bootsector (that is, starting at 0x7e00 ;)

## What if I wish to offer the user the option to boot several OSes ?

The easiest way to boot another OS is a mechanism called *chainloading*. Windows stores something akin to a second-stage bootloader in the boot sector of the *partition* it was installed in. When installing Linux, writing e.g. LILO or GRUB to the *partition* boot sector instead of the MBR is also an option. Now, the thing your MBR bootsector can do is to *relocate* itself (copying from 0x0000:0x7c00 to, traditionally, 0x0060:0x0000), parse the partition table, display some kind of menu and let the user chose which partition to boot from. Then, your (relocated) MBR bootsector would load that *partition* boot sector to 0x0000:0x7c00, and jump there. The partition boot sector would be none the wiser that there already was a bootsector loaded *before*, and could actually load yet *another* bootsector - which is why it's called *chainloading*. It doesn't really matter where you decide to relocate the boot sector as long as you don't overwrite the IVT (if IF in EFLAGS is set), the BDA or the EBDA.

You see that with displaying a menu in some intelligible way and accepting keystrokes, such a multi-option bootloader can get quite complex rather quickly. We didn't even touch the subject of booting from extended partitions, which would require sequentially reading and parsing multiple extended partition tables before printing the menu.

Taken to the extreme, boot managers like that can become as complex as a simple OS (much like GRUB is, which offers reading from various filesystems, booting Multiboot kernels, chainloading, loading initrd ramdisks etc. etc. - such internals will not be addressed here.

## How do I actually load bytes

BIOS interrupt 13h. Get info about it at Ralf Brown's Interrupt List, make sure you know floppies may fail one or two times, that you cannot read more than a track at once, and you're done. To read from the hard drive, you probably want int 13h, ah=0x42, drive number 0x80. Details in the interrupt list.

If you need guidance, feel free to check lowlevel.asm (http://clicker.cvs.sourceforge.net/clicker/c32-lxsdk/kernel/src/sosflppy/lowlevel.asm?view=log)

Note also that most File Systems involve some conversion between allocation units (blocks/clusters) and physical "Cylinder:Head:Sector" values. Those conversions are simple once you know the *sectors-per-track* and *heads* counts. Check out OSRC (http://www.nondot.org/sabre/os/articles) for additional info.

```
> Does anyone have a formula for converting DOS Sectors to
> Physical Sectors (Head, Cylinder, Sector) such as used in
> INT 13h?

DOS_sector_num = BIOS_sector_num - 1 + Head_num*Sectors_per_track
               + Track_num*Sectors_per_track*Total_heads

BIOS_sector_num = 1 + (DOS_sector_num MOD Sectors_per_track)
BIOS_Head_num   = (DOS_sector_num DIV Sectors_per_track) MOD Total_heads
BIOS_Track_num  = (DOS_sector_num DIV Sectors_per_track) DIV Total_heads
```

If you're loading above 1MB, you should proceed in 2 steps: first using BIOS to load in the "conventional" area, and then performing a `rep movsd` to place the data where they ultimately should go.

## Loaded. Gathering Information

The next step consist of collecting as much information as you can/need: amount of installed RAM, available video modes and things alike are easier to do in real mode, so better do them while in Real Mode than trying to come back to real mode for a trip later. Of course the exact requirements depend on your kernel.

A very simple solution here is to organize your information as a flat table (ala BIOS data area). An alternative could be to add those information as a structured flow: you keep an index at a well-known address (or at some address you'll pass to the kernel when loaded) and that index gives for each "key" the address of the corresponding data structure. E.g.

```
organization           lookup code (eax == signature)
+------+------+          mov esi, well_known_index_address
| RAM. | 1234 |        .loop:
| VBE. | 5678 |          cmp [esi],'END.'
| MODS | 9ABC |          je .notfound
| DISK | DEF0 |          add esi,8
| END. | ---- |          cmp [esi-4],eax
+------+------+          jne .loop
                         mov eax,[esi]
                         ret
```

## Ready. Entering Protected Mode ...

To enter protected mode you should first disable interrupts and set global descriptor table. After it set PE bit of CR0:

```
mov eax,cr0
or eax,1
mov cr0,eax
```

After it set registers and do a far jump to kernel. If data selector is 10h, code selector is 8 and kernel offset is 10000h do:

```
mov ax,10h
mov ds,ax
mov es,ax
mov fs,ax
mov gs,ax
mov ss,ax
jmp 8:10000h
```

Notes:

- that in this case, the GDT will be *temporary*. Indeed, the loader has no idea of what the kernel wants to do with the GDT, so all it can do is providing a minimal and let the kernel reload GDTR with an appropriate GDT later.
- it's common for the loader to keep interrupts disabled (the kernel will enable them later when an IDT is properly set up)

- give yourself the time about thinking whether you'll enable paging now or not. Keep in mind that debugging paging initialization code without the help of exception handlers may quickly become a nightmare !
- it is possible to perform more initialization once protected mode is enabled and before kernel is loaded. This will, however, require that you mix 16 bits and 32bits code in a single object file, which can quickly become a nightmare too...