Let's Start with the Bootloader

Introduction

When a computer powers on, a piece of code called bootloader is loaded and takes the control of the computer. Usually, the goal of the bootloader is loading the kernel of an operating system from the disk to the main memory and gives the kernel the control over the computer. The firmware of a computer is the program which loads the bootloader, in IBM-compatible computers the name of this program is BIOS (Basic Input/Output System).

There is a place in the hard disk called *boot sector*, it is the first sector of a hard disk [1], BIOS is going to load the content of the boot sector as a first step of running the operating system. Loading the boot sector's content means that BIOS reads the content from hard disk and loads it into the main memory (RAM). This loaded content from the boot sector should be the bootloader, and once its loaded into the main memory, the CPU will be able to execute it as any normal application we use in our computers. So, the last step performed by BIOS in this process is giving the control to the bootloader to do whatever it wants.

So, before start writing 539kernel, we need to write the bootloader that is going to load 539kernel from the disk. In x86 architecture, the size of the bootloader is limited to 512 bytes and due to this restricted size and the need of using low-level services, the bootloader is usually written in assembly language, also, because of this restricted size, we cannot depend on BIOS to load 539kernel because it going to be larger that 512 bytes, The reason of this limited size of the bootloader is because of the size of a sector in the hard disk itself. Each sector in the hard disk has the size of 512 bytes and as we have mentioned, BIOS is going to load the content of the first sector of hard disk, the boot sector, which, of course, as any sector its size is 512 bytes. Furthermore, the bootloader is going to run on an *x86 operating mode* known as *real mode* [2], what we need to know about that for now that under *real mode* is a 16-bit environment, so, even if the working processor is a 64-bit processor, we can only use 16-bits of available registers.

The booting process is too specific to the computer architecture as we have seen and it may differs between one architecture and another. Some readers, especially Computer Science students may notice that the academic textbooks of operating systems don't mention the boot loader or discuss it.

[1] A magnetic hard disk has multiple stacked *platters*, each platter is divided into multiple *tracks* and inside each track there are multiple *sectors*. The size of a sector is 512 bytes, and from here the restricted size of a boot loader came.

[2] Real Mode will be examined on the next chapter.

Hard Disk Structure

A hard disk consists of multiple *platters* which are stacked together one above the other, have you ever seen a CD or a DVD? A platter has exactly the same shape, refer to Figure @fig:a-platter. The both surfaces (top and down) of a platter are covered by a magnetic layer which stores the data. For each surface of a platter there is a read/write head on it, and as you guessed, the role of this head is to read from a surface or write to it, a head is attached to an arm. Those arms move horizontally, that is, back and forth, and because the other end of all of those arms are attached to the same physical part, they will be moved back and forth together to the same location at the same time. Figure @fig:platters-arms-heads shows how the platters, arms and read/write heads are assembled together.

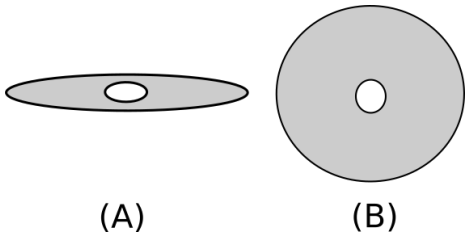

Figure 1: (A) Shows a platter when we see it from the side. (B) Shows a platter when we see it from top/down.

A surface of a platter is divided into a number of tracks and each track is divided into a number of sectors. In Figure @fig:tracks-sectors you can see how tracks and sectors are organized on a surface, the gray circles that have the same center (cocenteric) are the tracks, a track consists of a smaller parts which called sectors. A sector is the smallest unit that holds data in hard disks and as you know from our previous discussion, the first sector in a hard disk is the boot sector.

When the operating system needs to write some data to the hard disk or read from it, at least two mechanical moves [3] are performed. This first move is performed by the read/write arms, they move back or forth to be upon the track that contains the data we would like to read, this operation is known as *seek*. So, *seek time* is the time needed to put a specific track under read/write head. Until now, the read/write head is on the right track but also it is on a random sector [4], to reach the sector that we would like to read from (or write to) the platter rotates until the read/write head becomes on the desired sector. The speed of rotation is measured by the unit RPM (Revolutions Per Minute) and the needed time to reach the desired sector is known as *rotational latency*. Finally, the data will be *transferred* from the hard disk to the main memory, the time needed to transfer a number of bits known as *transfer time*.

[3] This fancy term "Mechanical Moves" means the physical parts of hard disk moves.
[4] Not exactly random, can you tell why?

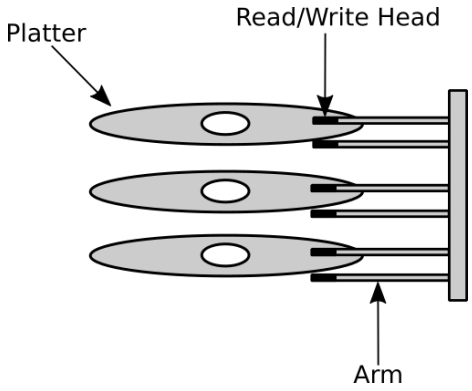

Figure 2: Shows how the parts of a hard disk are assembled together.

Let's say that a hard disk has 3 platters, which means it has 6 surfaces, arms and read/write head. When the operating system request from the hard disk to seek a specific track, for instance track #3, all 6 heads will seek the track #3 and when the seek operation ends, the 6 heads will point to track #3 on all 6 surfaces, that is, the top head of the first platter and the bottom head of it is going to point to track #3, and so on for the other 4 remaining heads, the collection of all these tracks that the heads point to at some point of time is called a *cylinder*.

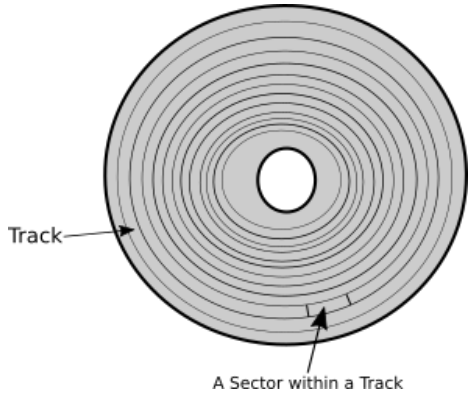

Figure 3: Shows Tracks and Sectors on a platter's surface.

Now, based on what we know about how a hard disk work, can we imagine what happens inside the hard disk when BIOS loads a bootloader? First, the arms will be *seek* the track number 0 [5], that is, the arms move back or forth until they reach the track #0, then the platter rotates until the read/write head become upon the sector #0, finally, the content of sector #0 is transferred to the main

[5] I didn't mention that previously, but yes, the bootloader resides in track #0.

memory.

BIOS Services

We are trying to write an operating system kernel, which means that our bootloader is running in a really harsh environment! Do you remember all libraries that we are lucky to have when developing normal software (user-space software), well, none of them are available right now! And they will not be available until we decide to make them so and work hard to do that. Even the simple function `printf` of C is not available.

But that's fine, for our luck, in this environment, where there is too little available for us to write our code, BIOS provides us with a bunch of useful services that we be can used in Real Mode, we can use these services in our bootloader to get things done.

BIOS services are like a group of functions in high-level languages that is provided by some library, each function does something useful and we deal with those functions as black boxes, we don't know what's inside these functions but we know what they do and how to use them. So, basically, BIOS provides us a library and we are going to use some of these functions in our bootloader.

BIOS services are divided into categories, there are video services category, disk services category, keyboard services category and so on and each category is labeled by a number called *interrupt number*. In high-level world, we witnessed the same concept but with different mechanism, for example, C standard library provides us with many services (functions) such as input/output functions, string manipulation functions, mathematical functions and so on, these functions are categorized and each category is label by the *library name*, for example, all input/output functions can be found in `stdio.h` and so on. In BIOS, for example, the category of video services has the interrupt number 10h [6].

Inside each services category, these is a bunch of services, each one can do a specific thing. Also, there are labeled by a number. In C, a service is a function labeled by a name (printf for example) and this function reside in a library (stdio.h for example) which is same as a category of services in BIOS. As we said, the interrupt number 10h represents the category of video services, and the service of printing a character on a screen is represented by the number 0Eh.

Interrupts is a fundamental concept in x86 architecture. What we need to know about them right now is that they are a way to call a specific code which is registered to them and calling an interrupt in assembly is really simple:

```
int 10h
```

[6] Note `h` in the number, that means this number is in hexadecimal numbering system. It does't equal the decimal number 10. When we use hexadecimal number we use `h` as a postfix.

That's it! We use the instruction `int` and gives it the interrupt that we would like to call as an operand. In this example, we are calling the interrupt 10h which is, as we mentioned multiple time, the category of BIOS video services. When the CPU executes this instruction, BIOS will be called and based on the interrupt number it will know that we want to use one of available video services, but which one exactly!

In the previous example, we actually didn't tell BIOS which video service we would like to use and to do that we need to specify service number in `ah` register before calling the interrupt.

```
mov ah, 0Eh
int 10h
```

That's it, all the BIOS services can be used in this exact way. First we need to know what is the interrupt number that the service which we which to use belong to, then, we need to know the number of the service that we're going to use, we put the service number in the register `ah` then we call the interrupt by its number by using `int` instruction. The previous code calls the service of printing a character on a screen, but is it complete yet? Actually no, we didn't specify what is the character that we would like to print. We need something like parameters in high-level languages to pass additional information for BIOS to be able to do its job. Well, lucky us! the registers are here to the rescue.

When a BIOS service need additional information, that is, parameters. It expects to find these information in a specific register. For example, the service 0Eh in interrupt 10h expects to find the character that the user wants to print in the register `al`. So, the register `al` is one of service 0Eh parameter. The following code requests from BIOS to print the character S on the screen:

```
mov ah, 0Eh
mov al, 'S'
int 10h
```

A Little Bit More of x86 Assembly and NASM

We need to learn a couple more things about x86 to be able to start. In NASM, each line in the source code has the following format.

```
label: instruction operands
```

The label is optional, the operands depend on x86 instruction in use, if it doesn't get any operand then we don't need to write any operand. To write comments on NASM we begin with semi-colon and write whatever we like after it as a comment, the rest of the source line will be considered as a part of the comment after writing the semi-colon.

A label is a way to give an instruction or a group of instructions a meaningful name, then we can use this name in other places in the source code to refer to

this instruction/group of instructions, we can use labels for example to call this group of instructions or to get the starting memory address of these instructions. Sometimes, we may use labels to make the code more readable.

We can say that a label is something like the name of a function or variable in C, as we know a variable name in C is a meaningful name that represents the memory address of the place in the main memory that contains the value of a variable, the same holds for a function name. Labels in NASM works in the same way, underhood it represents a memory address. The colon in label is also optional.

```
print_character_S_with_BIOS:
    mov ah, 0Eh
    mov al, 'S'
    int 10h
```

You can see in the code above, we gave a meaningful name `print_character_S_with_BIOS` for the bunch of instructions that prints the character `S` on the screen. After defining this label in our source code, we can use it anywhere in the same source code to refer to this bunch of instructions.

```
call_video_service int 10h
```

This is another example of labels. This time we eliminated the optional colon in label's name and the label here point to only one instruction. Please not that extra whitespaces and new lines doesn't matter in NASM, so, the following is equivalent to the one above.

```
call_video_service
    int 10h
```

Consider the following code, what do you think it does?

```
print_character_S_with_BIOS:
    mov ah, 0Eh
    mov al, 'S'

call_video_service:
    int 10h
```

Still it prints `S` on the screen. Introducing labels in the source code doesn't change its flow, it will be executed sequentially whether we used the labels or not. The sequence of execution will not be changed by merely using labels, if we need to change the sequence of execution we need to use other methods than labels. You already know one of these methods which is calling an interrupt.

So, we can say that labels are more general than a variable name or function name in C. A label is a human-readable name for a memory location which can contain anything, code or data!

Jump and Return Unconditionally

Let's start this section with a simple question. What happens when we call a function in C? Consider the following C code.

```
main()
{
    int result = sum( 5, 3 );

    printf( "%d\n", result );
}
```

Here, the function `main` called a function named `sum` here, this function reside in a different region in memory and by calling it we are telling the processor to go to this different region of memory and execute what's inside it, the function `sum` is going to do its job, and after that, in some magical way, the CPU is going to return to the original memory region where we called `sum` from and proceed the execution of the code that follows the calling of `sum`, in this case, the `printf` function. How does the CPU know where to return after completing the execution of `sum`?

The function which call another is named *caller* while the function which is called by the caller named *callee*, in the above C code, the caller is the function `main` while the callee is the function `sum`.

x86 Calling Convension

When a program is running, a copy of its machine code is loaded in the main memory, this machine code is a sequence of instructions which are understandable by the processor, these instructions are executed by the processor sequentially, that is, one after another in each cycle in the processor [7]. So, when a processor starts a new *instruction cycle*, it fetches the next instruction that should be executed from the main memory and executes it [8]. Each *memory location* in the main memory is represented and referred to by a unique *memory address*, that means each instruction in the machine code of the program under execution has a unique memory address, consider the following hypothetical example of the memory addresses of each instruction in the previous C code, note that the memory addresses in this example are by no means accurate.

```
100 main() {
110     int result = sum( 5, 3 );
120     printf( "%d\n", result );
```

[7] This is known as *von Neumann architecture* where both code and data are stored in the same memory and the processor uses this memory to read the instructions that should be executed, and manipulate the data which is stored in the same memory. There is another well-known architecture called *Harvard architecture* where the code and data are stored in two different memories. x86 uses *von Neumann architecture*.

[8] The instruction cycle is also called *fetch-decode-execute cycle*.

```
130 }

250 int sum( int firstNumber, int secondNumber ) {
260     return firstNumber + secondNumber;
270 }
```

So, the number on the left is the hypothetical memory address of the code line in the right, that means the function `main` starts from the memory address `100` and so on. Also, we can see that the callee `sum` resides in a far region of memory from the caller `main`.

Program Counter is a part of computer architecture which stores the *memory address* for the instruction that will be executed in the next instruction cycle of the processor. In x86, the program counter is a register known as *instruction pointer* and its name is `IP` in 16-bit and `EIP` in 32-bit.

When the above C code runs for the first time, the value of the instruction pointer will be `100`, that is, the memory address of the starting point of `main` function. When the instruction cycle starts it reads the value of the instruction pointer register `IP`/`EIP` which is `100`, it fetches the instruction which is stored in the memory location `100` and executes it [9], then the memory address of the next instruction `110` will be stored in the instruction pointer register for the next instruction cycle. When the processor starts to execute the instruction of the memory location `110`, this time, the value of `IP`/`EIP` will be `250` instead of `120` because, you know, we are calling the function `sum` which resides in the memory location `250`. Each running program has a *stack* which is a region of memory [10], we will examine later the *stack* in details but what is important for us now the following, when another function is called, in our case `sum`, the memory address of the next instruction of the callee `main` is *pushed* [11] into the stack, so the memory address `120` will be pushed into the stack before calling `sum`, this address is called *return address*. Now, assume that the processor is executing the instruction in the memory location *270*, that is, finishing the execution of the callee `sum`, after the execution, the processor will find the return address which is `120` in the stack, get it and put it in the register `IP`/`EIP` for the next instruction cycle [12]. So, this is the answer of our original question in the previous section "How does the CPU know where to return after completing the execution of `sum`?".

The Instructions `call` and `ret`

[9] For the simplicity of explanation, the details of *decoding* have been eliminated.

[10] The stack as a region of memory (x86 stack) is not same as the *data structure* stack, the former implements the later.

[11] Push means store something in a stack, this term is applicable for both x86 stack and the data structure stack, as we have said previously, x86 stack is an implementation of the stack data structure.

[12] External Reading: The is the cause of buffer overflow bugs.

The instruction `call` in assembly works exactly in the same way that we have explained in the previous section, it is used to call a code (or jump to a code) that resides in a given memory address. `call` pushes the return address into the stack, to return to the caller, the callee should use the instruction `ret` which gets [13] the return address from the stack and use it to resume the execution of the caller. Consider the following example.

```
call print_character_S_with_BIOS
call print_character_S_with_BIOS

print_character_S_with_BIOS:
    mov ah, 0Eh
    mov al, 'S'
    int 10h
    ret
```

You can see here that we have used the code sample `print_character_S_with_BIOS` to define something like C function by using the instructions `call` and `ret`. It should be obvious that this code prints the character `S` two times, as we have said previously, a label represents a memory address and `print_character_S_with_BIOS` is a label, the operand of `call` is the memory address of the code that we wish to call (or jump to), the instructions of `print_character_S_with_BIOS` will be executed sequentially until the processor reaches the instruction `ret`, at this point, the return address is obtained from the stack and the execution of the caller is resumed.

`call` performs an *unconditional jump*, that means processor will always jump to the callee, without any condition, later in this chapter we will see the instruction while performs a *conditional jump*, which only jumps to the callee when some condition is satisfied, otherwise, the execution of the caller is resumed.

The One-Way Unconditional Jump with The Instruction `jmp`

Like `call`, the instruction `jmp` jumps to the specified memory address, but unlike `call`, it doesn't store the return address in the stack which means `ret` cannot be used in the `callee` which is called by using `jmp`. We use `jmp` when we want to jump the a code that we will not return from it, `jmp` has the same functionality of `goto` statement in C. Consider the following example.

```
print_character_S_with_BIOS:
    mov ah, 0Eh
    mov al, 'S'
    jmp call_video_service

print_character_A_with_BIOS:
```

[13] Actually it *pop*s the value since we are talking about stack here.

```
    mov ah, 0Eh
    mov al, 'A'

call_video_service:
    int 10h
```

Can you guess what is the output? it is `S` and the code of the label `print_character_A_with_BIOS` will never be executed because of the line `jmp call_video_service` which is mentioned in the code of the label `print_character_S_with_BIOS`. If we remove the line of `jmp` from this code sample, both `S` and `A` will be printed on the screen. Another example which causes infinite loop.

```
infinite_loop:
    jmp infinite_loop
```

Comparison and Conditional Jump

In x86 there is a special register called *FLAGS* register [14]. It is the *status register* which holds the current status of the processor. Each usable bit of this register has its own purpose and name, and represents something different than any other bit on the same register, that is, each bit is a separate flag. For example, the first bit (bit 0) of FLAGS register is known as *Carry Flag* (CF) and the seventh bit (bit 6) is known as *Zero Flag* (ZF).

Many x86 instructions use FLAGS register to store their result on, one of those instruction is `cmp` which can be used to compare two integers, it takes to operands which are the two integers that we would like to compare then the processor stores the result in FLAGS register by using some mechanism that we will not mention here for the sake of simplicity. The following example compares the value which reside in the register `al` and `5`.

```
cmp al, 5
```

Now, let's say that we would like to jump a piece of code only if the value of `al` equals `5`, otherwise, the code of the caller continues without jumping. There are multiple instructions that perform *conditional* jump based on the result of `cmp`. One of these instructions is `je` which means *jump if equal*, that is, if the two operands of the `cmp` instruction equals each other, them jump to a specific code, another conditional jump instruction is `jne` which means *jump if not equal*, there are other conditional jump instruction and all of them named `Jcc` when they are discussed in Intel's official manual of x86. We can see that the conditional jump instructions have the same functionality of `if` statement in C. Consider the following example.

[14] In 32-bit x86 processors its name is *EFLAGS* and in 64-bit its name is *RFLAGS*.

```
main:
    cmp al, 5
    je the_value_equals_5
    ; The rest of the code of `main` label
```

This example jumps to the code of the label `the_value_equals_5` if the value of the register `al` equals `5`. In C, the above assembly example will be something like the following.

```
main()
{
    if ( register_al == 5 )
        the_value_equals_5();

    // The rest of the code
}
```

Like `jmp`, but unlike `call`, conditional jump instructions don't push the return address into the stack, which means the callee can't use `ret` to return and resume caller's code, that is, the jump will be *one way jump*. We can also imitate `while` loop by using `Jcc` instructions and `cmp`, the following example prints `S` five times by looping over the same bunch of code.

```
mov bx, 5

loop_start:
    cmp bx, 0
    je loop_end

    call print_character_S_with_BIOS

    dec bx

    jmp loop_start

loop_end:
    ; The code after loop
```

You should be familiar with the most of the code of this sample, first we assign the value `5` to the register `bx` [15], then we start the label `loop_start` which the first thing it does it comparing the value of `bx` with `0`, when `bx` equals `0` the code jumps to the label `loop_end` which contains the code after the loop. When `bx` doesn't equal `0` the label `print_character_S_with_BIOS` will be called to print `S` and return to the caller `loop_start`, after that the instruction `dec` is used to decrease `1` form its operand, that is `bx = bx - 1`, finally, the label `loop_start`

[15] Can you tell why we used `bx` instead of `ax`? [Hint: review the code of `print_character_S_with_BIOS`.]

will be called again and the code repeats until the value of `bx` reaches to `0`. The equivalent code in C is the following.

```
int bx = 5;

while ( bx != 0 )
{
    print_character_S_with_BIOS();
    bx--;
}

// The code after loop
```

Load String with The Instruction `lods`

It is well-known that `1` byte equals `8` bits. Moreover, there are two other size units in x86, a *word* which is `16` bits, that is, `2` bytes, and *doubleword* which is `32` bits, that is, `4` bytes. Some x86 instructions have multiple variants to deal with these different size units, while the functionality of an instruction is the same, the difference will be in the size of the data that a variant of instruction deals with. For example, the instruction `lods` has three variants `lodsb` which works a **b**yte, `lodsw` which works with a **w**ord and `loadsd` which works with a **d**oubleword.

To simplify the explanation let's consider `lodsb` which works with a single byte, its functionality is too simple, it reads the value of the register `si`, it deals with it as a memory address and transfers a byte from the content of memory address to the register `al`, finally, it increments the value of `si` by `1` byte. The same holds for the other variants of `lods`, only the size of the data, the used registers and the increment size are different, the register which is used in `lodsw` is `ax` [16] and `si` is incremented by `2` bytes, while `lodsd` uses the register `eax` [17] and `si` is incremented by `4` bytes. [18]

[16] Because the size of `ax` is a **word**

[17] Because the size of `eax` is a **doubleword**.

[18] As fun exercise, try to figure out why are we explaining the instruction `lodsb` in this chapter, what is the relation between this instruction and the bootloader that we are going to write? Hint: Review the code of `print_character_S_with_BIOS` and how to print a character by using BIOS services. If you can't figure the answer out don't worry, you will get it soon.

NASM's Pseudoinstructions

When you encounter the prefix [19] *pseudo* before a word, you should know that describes something fake, false or not real [20]. NASM provides us a number of **Pseudo**instructions, that is, they are not real x86 instructions, the processor doesn't understand them and they can't be used in other assemblers [21], on the other hand, NASM understands those instructions and can translate them to something understandable by the processor. They are useful, and we are going to use them to the writing of the bootloader easier.

Declaring Initialized Data

The concept of *declaring something* is well-known by the programmers, In C for example, when you *declare* a function, you are announcing that this function *exists*, it is there, it has a specific name and takes the declared number of parameters [22]. The same concept holds when you declare a variable, you are letting the rest of the code know that there exists a variable with a specific name. When we declare a variable, without assigning any value to it, we say that this variable is *uninitialized*, that is, no initial value has been assigned to this variable when it is declared, later on a value will be assigned to the variable, but not as early of its declaration. In contrast, a variable is *initialized* when a value is assigned to it when it's declared.

The pseudoinstructions `db`, `dw`, `dd`, `dq`, `dt`, `ddq`, and `do` helps us to declare initialized *data*, and with using *labels* when can mimic the concept of *initialized variable* in C. Let's consider `db` as an example, the second letter of `db` means *b*ytes, that means `db` declare and initialize a byte of data.

```
db 'a'
```

The above example reserve a byte in the memory, this is the declaration step, then the character `a` will be stored on this reserved byte of the memory, which is the initialization step.

```
db 'a', 'b', 'c'
```

In the above example we have used comma to declare three bytes and store the values `a`, `b` and `c` respectively on them, also, on memory these values will be

[19] In linguistics, which is the science that studies languages, a prefix is a word (actually a morpheme) that is attached in the beginning of another word and changes its meaning, for example, in **un**do, **un** is a prefix.

[20] For example, in algorithm design which is a branch of computer science, the word **pseudo**code which means a code that is written in a fake programming language. Another example is the word **pseudo**science, a statement is a pseudoscience when it is claimed to be a scientific fact, but in reality it is not, that is, it doesn't follow the scientific method.

[21] Unless, of course, they are provided in the other assembler as pseudoinstructions.

[22] It is important to note that *declaring* a function in C differs from *defining* a function, the following declares a function: `int foo();` You can see that the code block (the source doe) of `foo` is not a part of the declaration, once the code block of the function is presented, we say this is the *definition* of the function.

stored *contiguously*, that is, one after another, the memory location [23] of the value `b` will be right after the memory location of value `a` and the same rule for `c`. Since `a`, `b` and `c` are of the same type, a character, we can write the previous code as the following and it gives as the same result.

```
db 'abc'
```

Also, we can declare different types of data in the same source line, given the above code, let's say that we would like to store the number `0` after the character `c`, this can be achieved by simply using a comma.

```
db 'abc', 0
```

Now, to make this data accessible from other parts of the code, we can use a label to represent the starting memory address of this data. Consider the following example which define the label `our_variable`, after that, we can use this label to refer to the initialized data.

```
our_variable db 'abc', 0
```

Repeating with `times`

To repeat some source line multiple times, we can use the pseudoinstruction `times` which takes the number of desired repetitions as first operand and the instruction that we would like to execute repeatedly as second operand. The following example prints `S` five times on the screen.

```
times 5 call print_character_S_with_BIOS
```

Not only normal x86 instructions can be used with `times`, also NASM's pseudoinstructions can be used with `times`. The following example reserves `100` bytes of the memory and fills them with `0`.

```
times 100 db 0
```

NASM's Special Expressions `$` and `$$`

In programming languages, an *expression* is a part in the code that evaluates a value, for example, `x + 1` is an expression, also, `x == 5` is an expression. On the other hands, a *statement* is a part of the code that performs some actions, for example, in C, `x = 15 * y;` is a statement that assigns the values of an expression to the variable `x`.

NASM has two special expressions, the first one is `$` which points to the beginning of the *assembly position* of the current source line. So, one ways of implementing infinite loop is the following: `jmp $`. The second special expression is `$$` which points to the beginning of the current *section*.

[23] Hence, the memory address.

The Bootloader

As you have learned previously, the size of the bootloader should be `512` bytes, the firmware loads the bootloader in the memory address `07C0h`, also, the firmware can only recognize the data in the first sector as a bootloader when the data finishes with the magic code `AA55h`. When 539kernel's bootloader starts, it shows two messages for the user, the first one is "The Bootloader of 539kernel." and the second one "The kernel is loading...", after that, it is going to read the disk to find 539kernel and loads it to memory, after loading 539kernel to memory, the bootloader gives the control to the kernel by jumping to the start code of the kernel. Till this point, 539kernel doesn't exist, we haven't write it yet, instead of loading 539kernel, the bootloader is going to load a code that prints "Hello World!, From Simple Assembly 539kernel!". In this section, we are going to write two assembly files, the bootloader `bootstrap.asm` and `simple_kernel.asm` which is the temporary replacement of 539kernel, also, `Makefile` which complies the source code will be presented in this section.

Implementing the Bootloader in `bootstrap.asm`

Till now, you have learned enough to understand the most of the bootloader that we are going to implement, however, some details have not been explained in this chapter and have been delayed to be explained later. The first couple lines of the bootloader is an example of not explained concepts, our bootloader source code starts with the following.

```
start:
    mov ax, 07C0h
    mov ds, ax
```

First, we define a label named `start`, there is no practical reason to define this label [24], the only reason of defining it is the readability of the code, when someone else tries to read the code, it should be obvious for her that `start` is the starting point of executing the bootloader.

The job of next two lines is obvious, we are moving the hexadecimal number `07C0` to the register `ax` to be able to move it to the register `ds`, note that we can't store the value `07C0` directly in `ds` by using `mov` as the following: `mov ds, 07C0h`, due to that, we have put the value on `ax` and then moved it to `ds`, so, our goal was to set the value `07C0` in the register `ds`. Now, you may ask why we want the value `07C0` in the register `ds`, this is story for another chapter, just take these two lines on faith, and you will know later the purpose of them. Let's continue.

```
    mov si, title_string
    call message_string
```

[24] Such as jump to it for example.

```
    mov si, message_string
    call print_string
```

This block of code prints the two messages, both of them are represented by a separate label `title_string` and `message_string`, you can see that we are calling the code of the label (or the function) `print_string`, its name indicates that it prints a *string* of character, and you can infer that the function `print_string` receives the address of the string that we would like to print as a parameter in the register `si`, the implementation of `print_string` will be examined in a minute.

```
    call load_kernel_from_disk
    jmp 0900h:0000
```

These two lines represent the most important part of any bootloader, first a function named `load_kernel_from_disk` is called, we are going to define this function in a moment, as you can see from its name, it is going to load the code of the kernel from disk into the main memory and this is the first step that make the kernel able to take the control over the system. When this function finishes its job and returns, a jump is performed to the memory address `0900h:000`, but before discussing the purpose of this line let's define the function `load_kernel_from_disk`.

```
load_kernel_from_disk:
    mov ax, 0900h
    mov es, ax
```

These couple of lines, also, should be taken on faith. You can see, we are setting the value `0900h` on the register `es`. Now, we move to the most important part of this function.

```
    mov ah, 02h
    mov al, 01h
    mov ch, 0h
    mov cl, 02h
    mov dh, 0h
    mov dl, 80h
    mov bx, 0h
    int 13h

    jc kernel_load_error

    ret
```

This block of code *loads* the kernel from the disk into the memory and to do that it uses the BIOS Service `13h` which provides services that perform operations of the disks such as reading and writing. The service number which is `02h` is specified on the register `ah`, this service read sectors from disks and loads them

into the memory, the following registers should be set when we use the service `02h` in `13h`:

The value of register `al` is the number of sectors that we would like to read, in our case, we read only `1` sector, the size of our temporary kernel `simple_kernel.asm` doesn't exceed `512` bytes. Please keep in mind that we are going to store our kernel right after the bootloader in the disk, knowing that, you can make sense of the registers' values `ch`, `cl` and `dh` that we will be explained next. The value of register `ch` is the track number we would like to read from, in our case, it is the track `0`. The values of the register `cl` is the sector number that we would like to read its content, in our case, it is the second sector. The value of the register `dh` is the number of head. The values of of `dl` specifies which disk we would like to read from, the value `0h` in this register means that we would like to read the sector from a floppy disk, while the value `80h` means we would like to read from the hard disk `#0` and `81h` for hard disk `#1`, in our case, the kernel is stored in the hard disk `#0`, so, the value of `dl` should be `80h`. Finally, the value of the register `bx` is the memory address that the content will be loaded to, in our case, we are reading one sector, and its content will be stored on the memory address `0h` [25].

When the content is loaded successfully, the BIOS Service `13h:02h` is going to set the carry flag to `0`, otherwise, it sets the carry flag to `1` and stores the error code in register `ax`, the instruction `jc` is a conditional jump instruction that jumps when `CF = 1`, that is, when the value of the carry flag is `1`. That means our bootloader is going to jump to the label `kernel_load_error` when the kernel isn't loaded correctly.

If the kernel is loaded correctly, the function `load_kernel_from_disk` returns by using the instruction `ret` which makes the processor to resume the main code of our bootloader and executes that instruction which is after `call load_kernel_from_disk`, this instruction is instruction `jmp 0900h:0000` which gives the control to the kernel by jumping to its starting point, that is, the memory location where we loaded our kernel in. This time, the operand of `jmp` is an *explicit* memory address `0900h:0000`, it has two parts, the first part is the one before the colon, you can see that it is the same value that we have loaded in the register `es` in the beginning of `load_kernel_from_disk` function. The second part of the memory address is the one after the colon, it is `0h` [26] which is the *offset* that we have specified in the register `bx` in `load_kernel_from_disk` before calling `02h:13h`, the both parts combined represent the memory address that we have loaded our kernel into and the details of the two parts of this memory address will be discussed in chapter .

Now we have finished the basic code of the bootloader, we can start defining that labels that we have used before in its code. We start with the label `kernel_load_error` which simply prints an error message that is stored on

[25] Not exactly the memory address 0h. We will see why in the next chapter.
[26] Here, `0h` is equivalent to `0000`.

another label, as we have done previously, the label `kernel_load_error` prints the error message by using the function `print_string`, after printing the message, nothing can be done, so, `kernel_load_error` enters an infinite loop.

```
kernel_load_error:
    mov si, load_error_string
    call print_string

    jmp $
```

Our previous samples of using the BIOS Service `0Eh:10h` were printing only one character, in real world, we need to print a *string* of characters and that's what the function `print_string` exactly does, it takes the memory address which is stored in the register `si` and prints the character which is stored in it, then it goes to the next memory address and prints the character which is stored in it and so on, that is, `print_string` prints a string character by character by using loop. So, you may ask, how `print_string` can know when should it stop? There should be a stop condition in any loop, otherwise, it is an infinite loop.

A string in C programming language, as in our situation, is an *array of characters*, and the same problem of "where does a string end" is encountered in C programming language, to solve the problem, each string in C programming language ends with a special character named *null character* and represented by the symbol `\0` in C [27], so, you can handle any string in C character by character and once you encounter the null character `\0` that means you have reached the end of the string. We are going to use the same mechanism in our bootloader to recognize the end of a string by putting the value `0` as a marker in the end of the string. By using this way, we can now use the service `0Eh:10h` to print any string character by character through a loop and once we encounter the value `0` we can stop the printing.

```
print_string:
    mov ah, 0Eh

print_char:
    lodsb

    cmp al, 0
    je printing_finished

    int 10h

    jmp print_char

printing_finished:
    mov al, 10d ; Print new line
```

[27] This type of strings named *null-terminated strings*.

```
    int 10h

    ; Reading current cursor position
    mov ah, 03h
    mov bh, 0
    int 10h

    ; Move the cursor to the beginning
    mov ah, 02h
    mov dl, 0
    int 10h

    ret
```

When `print_string` starts, the BIOS services number `0Eh` is loaded in `ah`, this operation need to take a place just one time for each call for `print_string`, so it is not a part of the next label `print_char` which is a part of `print_string` and it will be executed after moving `0Eh` to `ah`.

As you can remember, that parameter of `print_string` is the memory address which contains the string that we would like to print, this parameter is passed to `print_string` via the register `si`, so, the first thing `print_char` does is using the instruction `lodsb` which is going to transfer the first character of the string to the register `al` and increase the value of `si` by `1` byte, after that, we check the character that has been transferred from the memory, if it is `0`, that means we have reached to the end of the string and the code jumps to the label `printing_finished`, otherwise, the interrupt `10h` of BIOS is called to print the content of the register `al` on the screen, then we jump to `print_char` again to repeat this operation until we read the end of the string. When printing a string finishes, the label `printing_finished` starts by printing a new line after the string, the new line is represented by the number `10` in ASCII after that we are going to use the service `03h` to read the current position of the cursor, then we use the service `02h` to set the cursor to position `0` by passing it to the register `dl`, otherwise, the messages in the new lines will be printed in the position where the previous string finished, finally the function returns to the caller by using the instruction `ret`.

```
title_string        db  'The Bootloader of 539kernel.', 0
message_string      db  'The kernel is loading...', 0
load_error_string   db  'The kernel cannot be loaded', 0
```

The code above defines the strings that have been used previously in the source code, note the last part of each string which is the null character that indicates the end of a string [28].

Now, we have written our bootloader and the last thing to do is to put the *magic*

[28] Exercise: What will be the behavior of the bootloader if we remove the null character from `title_string` and `message_string` and keeps it in `load_error_string`?

code in the end of it, the magic code which is a `2` bytes value should be the last two bytes in the first sector, that is, in the locations `510` and `511` [29], otherwise, the firmware will not recognize the content of the sector as a bootloader. To ensure that the magic code is written on the correct location, we are going to fill the empty space between the last part of bootloader code and the magic code by zeros, this can be achieved by this line.

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

So, the instruction `db` will be called `510-($-$$)` times, this expression gives us the remaining empty space in our bootloader before the magic code, and because the magic code is a `2` bytes value we subtract `($-$$)` from `510` instead of `512`, we will use these two bytes for the magic code, the expression `($-$$)` uses the special expressions of NASM `$` and `$$` and it gives the size of the bootloader code until this line. Finally, the magic code is presented.

```
dw 0xAA55
```

Implementing `simple_kernel.asm` and Makefile

The `simple_kernel.asm` which the bootloader loads is too simple, it prints the message "Hello World!, From Simple Assembly 539kernel!", we don't need to go through its code in details since you know most of it.

```
start:
    mov ax, cs
    mov ds, ax

    ; --- ;

    mov si, hello_string
    call print_string

    jmp $

print_string:
    mov ah, 0Eh

print_char:
    lodsb

    cmp al, 0
    je done

    int 10h
```

[29] The location number starts from `0`.

```
    jmp print_char

done:
    ret

hello_string db 'Hello World!, From Simple Assembly 539kernel!', 0
```

The only lines that you are not familiar with until now are the first two lines in the label `start` which will be explained in details in the next chapter. Finally the `Makefile` is the following.

```
ASM = nasm
BOOTSTRAP_FILE = bootstrap.asm
KERNEL_FILE = simple_kernel.asm

build: $(BOOTSTRAP_FILE) $(KERNEL_FILE)
    $(ASM) -f bin $(BOOTSTRAP_FILE) -o bootstrap.o
    $(ASM) -f bin $(KERNEL_FILE) -o kernel.o
    dd if=bootstrap.o of=kernel.img
    dd seek=1 conv=sync if=kernel.o of=kernel.img bs=512
    qemu-system-x86_64 -s kernel.img

clean:
    rm -f *.o
```