Developing

A Computer Operating System From Scratch

An Attempt To Introduce OS Development At Beginner Level

TINU TOM

Developing A Computer Operating System From Scratch

An Attempt To Introduce OS Development At Beginner Level

TINU TOM

This book is for sale at http://leanpub.com/OS-DEV

This version was published on 2021-07-28

* * * * *

* * * * *

Table of Contents

Preface

We all have used an operating system, whether you are using a mobile phone or a computer or any electronic devie. All of this devices have software that works close to the hardware.

Most of the operating system are split into two main components which are the kernel and the shell. Kernel of an operating system is what communicates with the hardware, manages memory etc and do all other main stuff etc… The main aim of the kernel is to provide an abstration level to the shell of that operating system which is the software that the user directly communicates with. Shell of an operating system is what you basically see on the screen and what you directly interact with. When you commands the shell to do a specific thing such as creating a file, the shell request for that service to the kernel and kernel does all of the things needed to create a file. So basically shell is only a middle stander between you and your device.

Let's learn these basic theories later, Let me say the reason which led me to write this book. OS develoment is a topic which is seen at an angle so that most people think it can be done only by a small section of geeks. The reason they say is that "Low level stuff is hard to learn and only the brighest ones could do that". And that argument is totally wrong as there are many people including children and teenagers who do the electronic stuff(Including programming), These people will only get the smallest level of abstraction when developing their projects and all other low level stuff should be done their own.

But when coming to the side of creating an operating system(Specifically on `x86` architecture), People say that it's too hard, but it's not.

Your question now may be "Then why does only a small section of people do that", and the answer of that question is that "There is only a small level of documentation for beginners to get started", According to the current situation, People who are not too old in software development area could not

do this things as all of the documentation are hard to understand for beginners.

Also currently there is no book giving a complete guide to this area apart from some websites which is meant for the “Too old people”.

I will promise you that, even if you are starting in this field or know only some part of this area, you could read this book. We will start learning from programming in c(Which is a basic need for os development), and assembly, and then forward, we will dive into real os development on the `intel's x86` architecture which is where operating systems such as windows and linux mainly run on.

So for now, you need to take away all things in your mind about os development and focus mainly on learning. I will explain each and every section deeply and also explain how the code given in this book works fully.

Getting Started

Learning os development is fun, But before starting, we need some tools which are essential for development and testing. I will explain you fully how to set everything to get started working on the project. We will use Windows as operating system to develop our project, But the methods discussed in this book can also be implemented in Linux. There is nothing stopping you from doing that.

Installing QEMU

To run the operating system we develop, we have a total of three options.

1. Use an emulator
2. Use a virtual machine
3. And finally, Running it on the the real hardware(By booting from a cd or usb)

Using an emulator or virtual machine works somewhat alike(From an outside perspective). But they are not the same(From an internal perspective) which is why they are called different as emulators and virtual machines.

Virtual machine softwares are used to run software which the processor could directly execute to a great extend. But the purpose of emulators is so that it could enable us running software which is intended to run on different architecures, but also it could emulate the architecture which it is currenly running on.

The final option to run the os we develop is to run in the real hardware, by burning the raw binary to a cd or usb and booting it on the real hardware. If you want to run the os we develop on the real hardware, please ensure to disable Secure Boot option provided in latest computers from bios. We don't really encourage you running the os in real hardware unless you clearly know what you are doing. This is because of many reasons one of which is that you could corrupt your hard disk if you program the os incorrectly, So what you

need to do is, First try running it on an emulator or virtual machine and study it's behaviour and when you confirm it works safe, You could try running on real hardware.

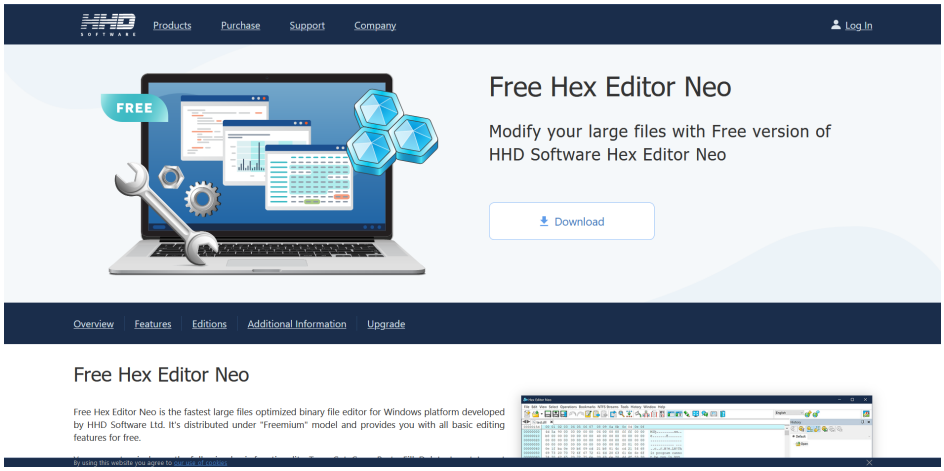
Please note that i will not be responsible for any damages you do to your system by using unsafe approches in any manner. If you want to run the os on real hardware, you could try running it on a secondary system which you won't mind if it gets harmed.

For now, We will try running the os that we are going to develop on an Emulator named QEMU. You could find in on `https://www.qemu.org/`

The installation of qemu is straight forward and you could carry it your own.

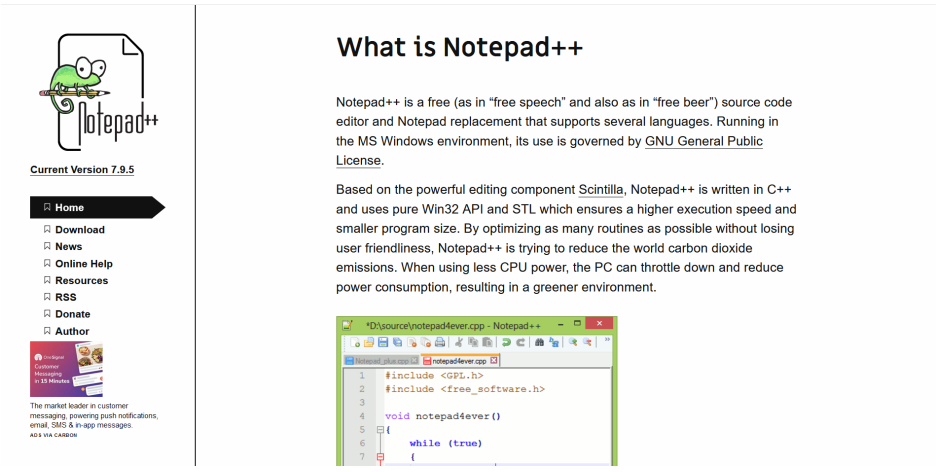
Installing A Hex Editor

You could install any Hex Editor in your machine if You know how to use it. But i will follow with a hex editor named Hex Editor Neo found at `https://www.hhdsoftware.com/free-hex-editor`. If you don't know what a hex editor is, we will discuss it in later chapters but for now, you could download it.



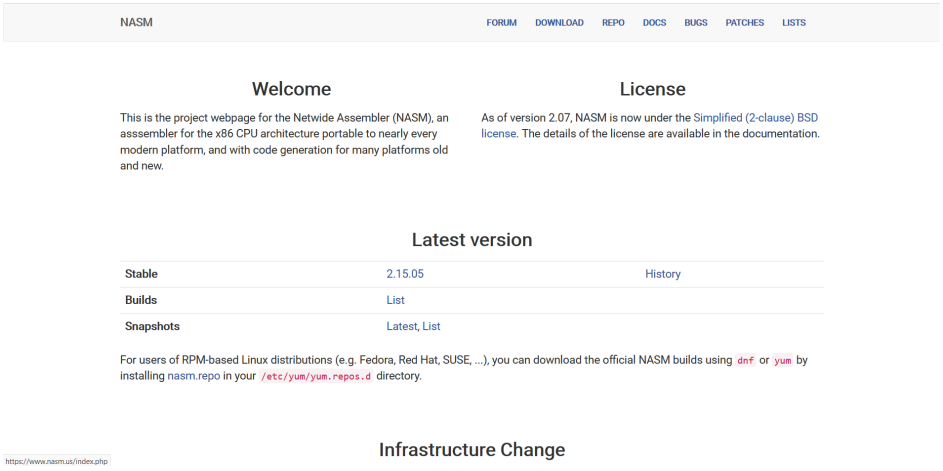
Installing Notepad++

We will use Notepad++ as our ide to develop the project. You could find in on `https://notepad-plus-plus.org/`



Installing NASM

Nasm is a widely used assembler. We will learn what an assembler is, Why and how to use it and every point you need to know to get started later. You could find it on `https://www.nasm.us/`

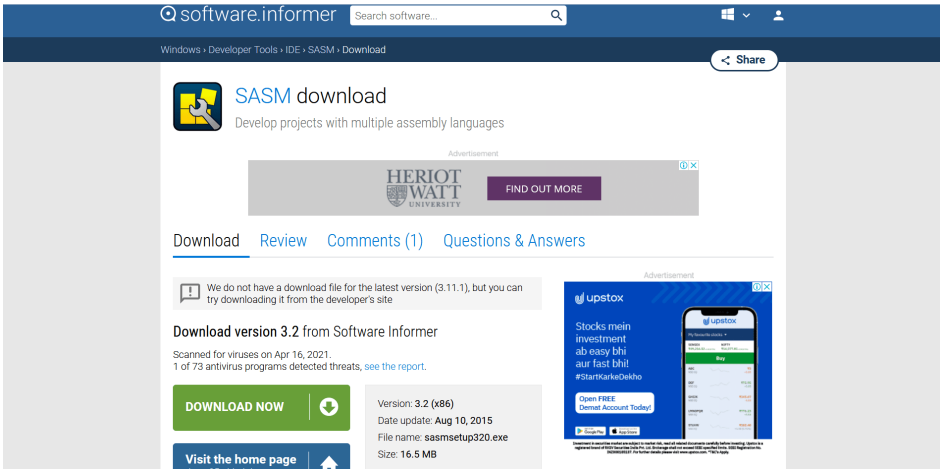


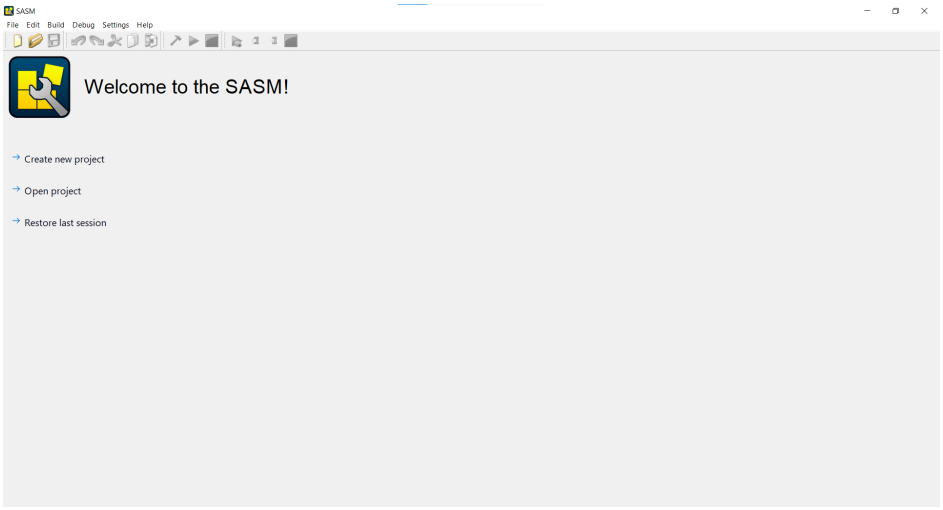
Installing SASM

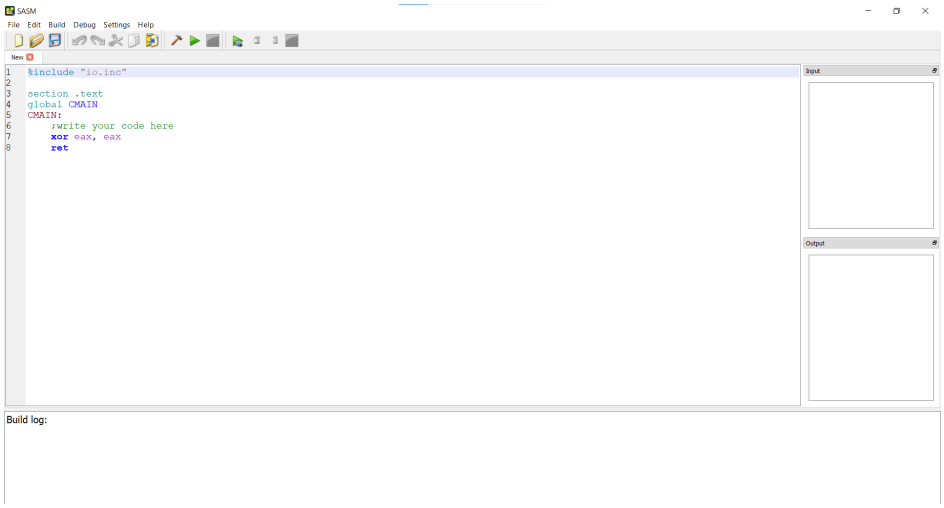
SASM is an ide which helps build assembly programs easily. There is no special need to download sasm for programming in assembly as we have already discussed installing nasm as an assembler, But for now for the sake of simplicity in learning developing applications in assembly, we could use this. If you currently know developing assembly programs, You could skip this step and also skip the chapter teaching programming in assembly.

By using sasm, You can avoid learning the concept of linking for some time and it also provides functions to print strings and numbers easily, Else you need to write your own routine or download special ones to print to screen. And the main part is that you could run and test your programs in one windows. You could download it from

`https://sasm.software.informer.com/download/`


software.informer
Search software...
Windows › Developer Tools › IDE › SASM › Download
Share
SASM download
Develop projects with multiple assembly languages
Advertisement
HERIOT WATT UNIVERSITY
FIND OUT MORE
Download
Review
Comments (1)
Questions & Answers
We do not have a download file for the latest version (3.11.1), but you can try downloading it from the developer's site
Download version 3.2 from Software Informer
Scanned for viruses on Apr 16, 2021.
1 of 73 antivirus programs detected threats, see the report.
DOWNLOAD NOW
Visit the home page
Version: 3.2 (x86)
Date update: Aug 10, 2015
File name: sasmsetup320.exe
Size: 16.5 MB
Advertisement
upstox
Stocks mein investment ab easy bhi aur fast bhi!
#StartKarkeDekho
Open FREE Demat Account Today!

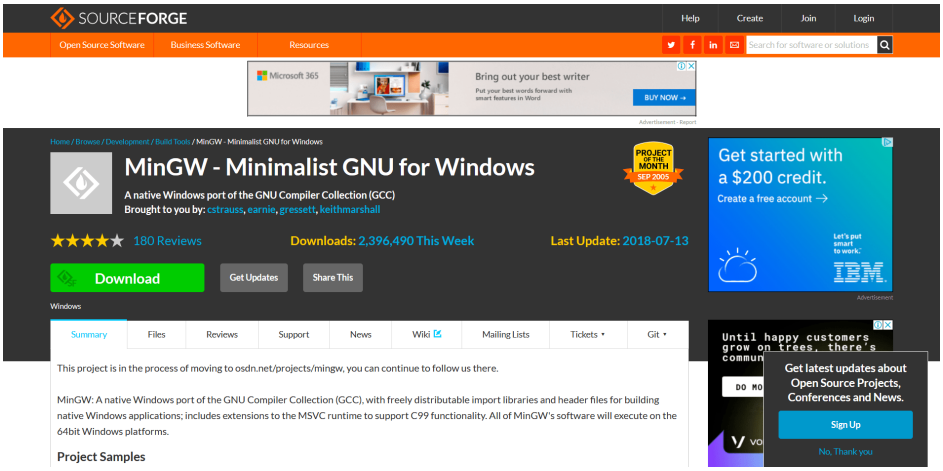

SASM
File Edit Build Debug Settings Help
Welcome to the SASM!
Create new project
Open project
Restore last session



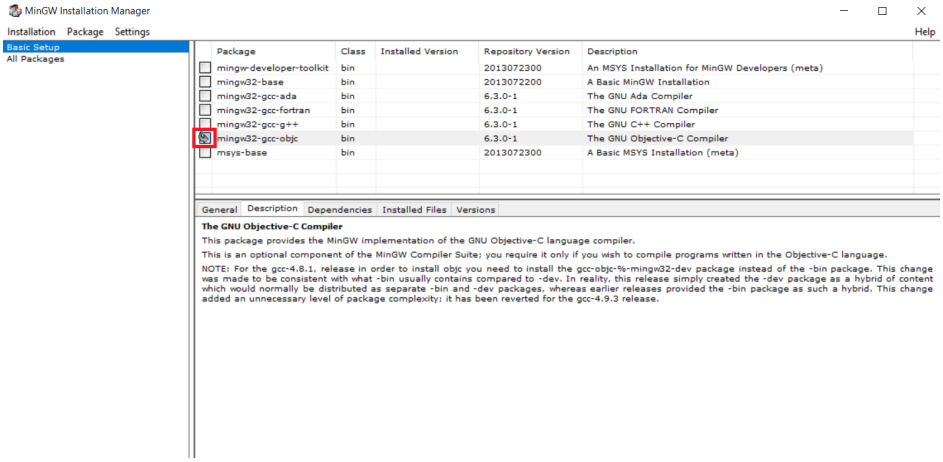
Installing MinGw For Compiling C Programs

MinGw is a collection of tools which could be used to develop application in windows platform. We will use the c compiler provided by MinGw to develop our os. C is a mid level language best suited to speed up os development. C was initially developed to develop the unix operating system, We could use great features provided by c to get our os running. If you are new to Programming in c, we will learn necessary thing in upcoming chapters. I will only use the methods that you could easily follow to develop the os in c. If you currently know how to program in c and/or assembly, You could catch os development faster, but others don't need to worry, i will carry you along with the journey.

To install MinGw, you need to go to
`https://sourceforge.net/projects/mingw/`



Check the Following option after starting the installer:



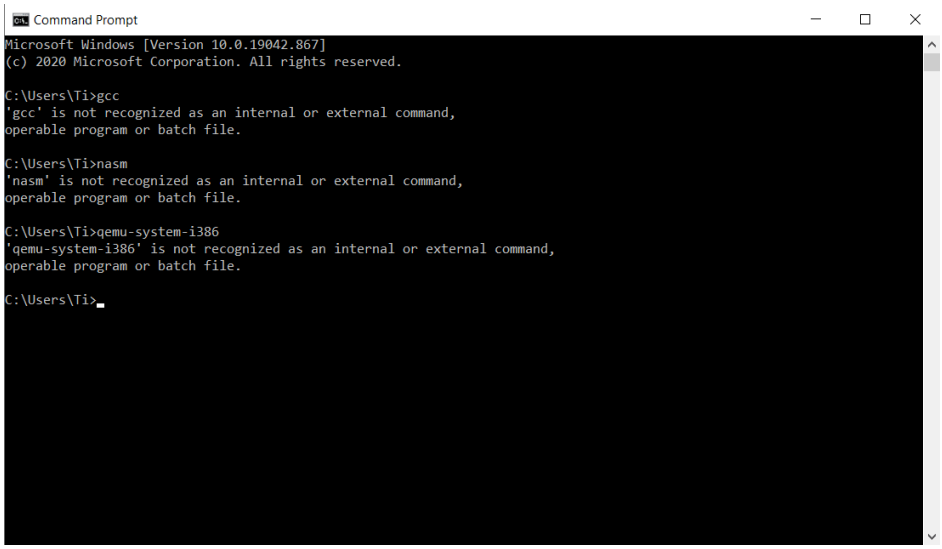
And finally, Install it.

Adding The Downloaded Softwares To Environment Path

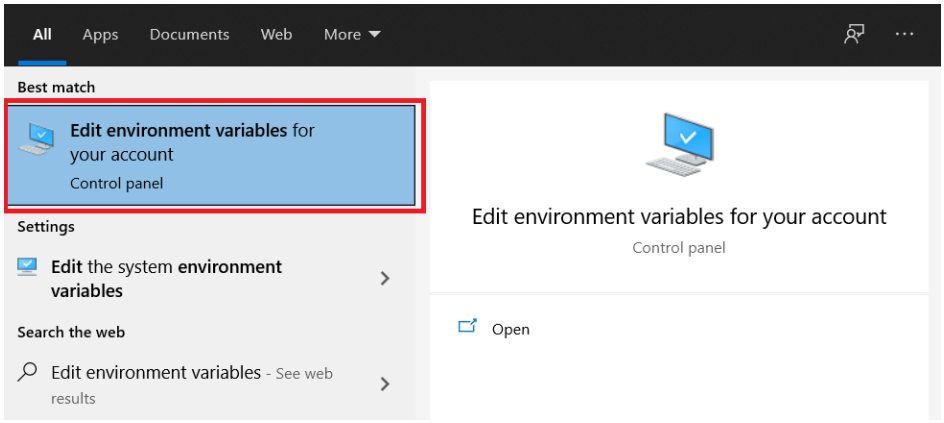
Finally, We have downloaded all tools that we need to get started in this learning process. But we need to access three of these softwares from the command line(`cmd.exe`).

For that, go to command prompt by typing cmd in the search bar and type `gcc` and hit enter. You will get an error message. Also type `nasm` and `qemu-`

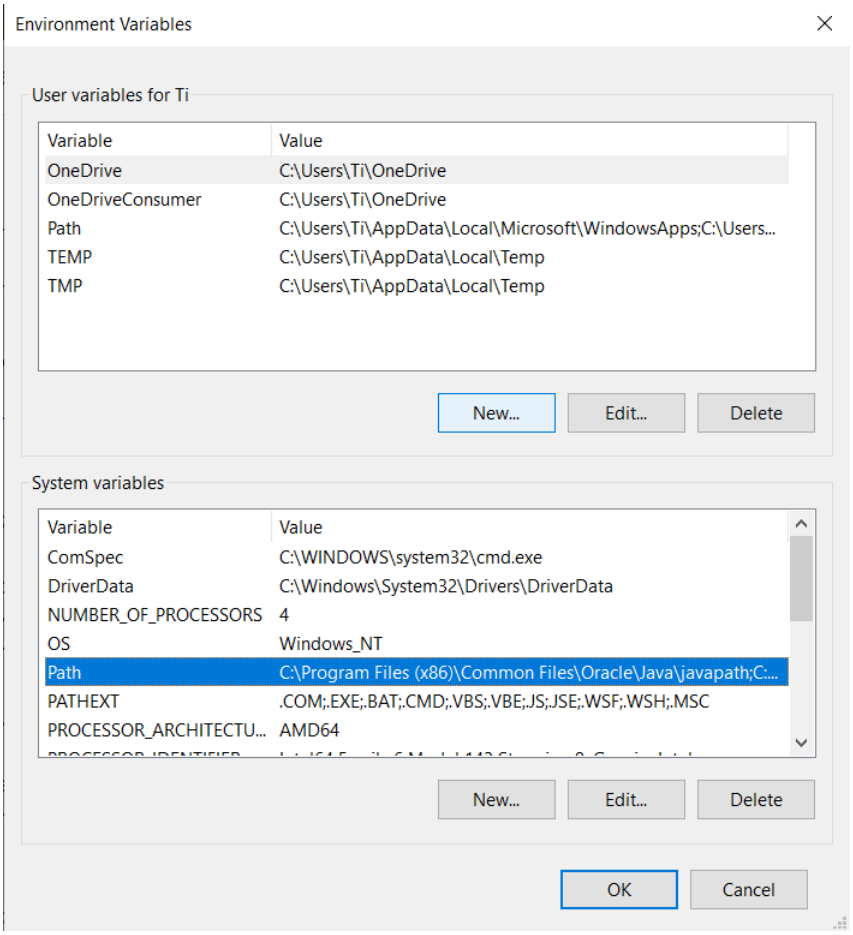
`system-i386` in it. It Also will show an error message.



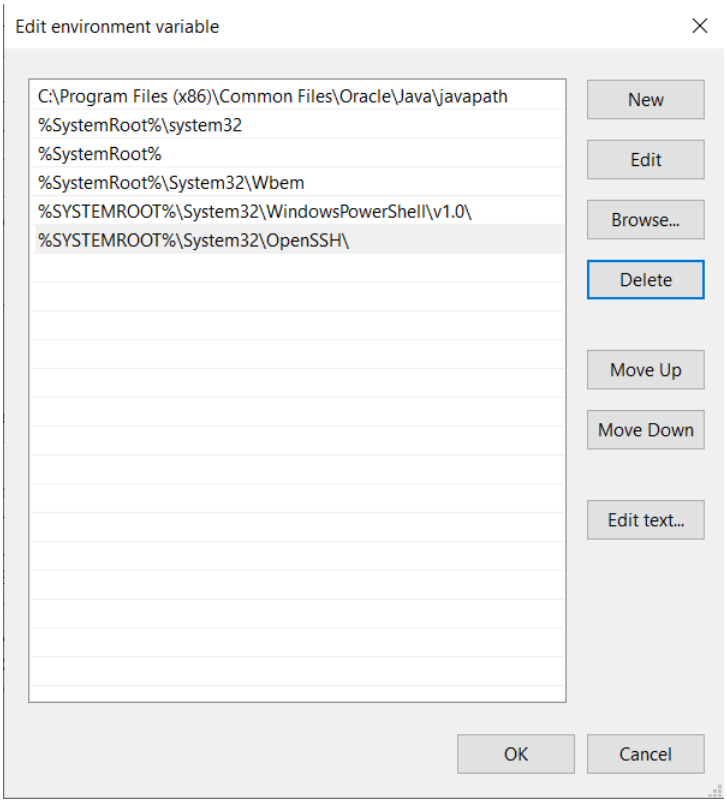
We need to access these softwares by typing it's name in the command prompt, to do that, search 'Edit environment variables' in the search box and select the first option



Now select 'Environment Variables…' button in the 'Advanced' tab,
In the new window, click on Path in the System variables section and click edit



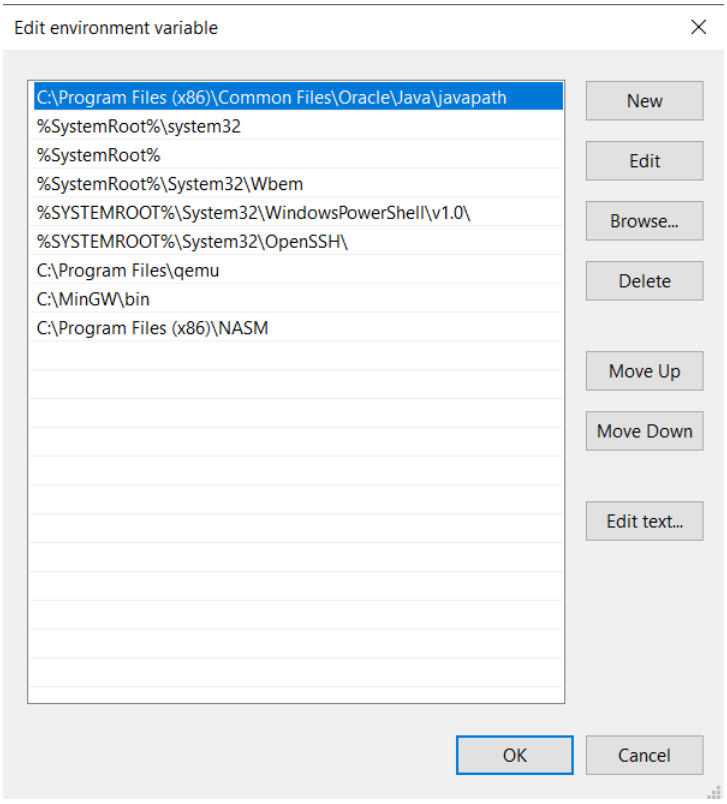
You will now get a window like the one below(Please note that all system may not be same like this as different systems could have different settings, but it will look somewhat like this):



Now you need to add the location to the bin folder of three of the softwares that we downloaded by clicking New button and pasting it there. You need to set the location of Qemu , Nasm and MinGw. In My system the following is the location to those folders.

1)Qemu : C:\Program Files\qemu
2)MinGw : C:\MinGW\bin
3)Nasm : C:\Program Files (x86)\NASM

Now, After adding all these locations, the window looks somewhat like this(Please not that, the location of these application i show here might not be the case for you, It will depend on the settings you adjusted when installing those softwares):



Now close all of the existing windows only by pressing OK.

Now you could type `gcc`, `nasm` and `qemu-system-i386` in the cmd and you will get some messages(Probably like 'Input file missing' , but thats ok)

We are now ready to start gettings our hands dirty by doing , LETS GOOOO!!!!!!

Programming In C

Introduction

`C` is generally called as a high level language, but some of them consider it as a mid level language. The reason being that it do not provide as lot of abstraction which is generally given by languages such as `java`, `python` and other new trendy languages.

There is a great chance for your application to get crashed probably because of memory corruption or wrong system calls etc when using c. If you are new to c, don't be scared as everything can be learned by doing. Learning advanced c will make you more knowledgable about the working of computers. In this book, we will only use very basic concepts of c, We will not use any advanced concepts and will try to keep it the simplest as possible. This is so that, by using this simple methods , there is a great chance for you to not get demotivated.

A common misconception about c is that, it is a program which converts human readable source code to binary. But c is only a programming standard which defines how the compiler should accept instructions. What c does is only translate your source code to it's assembly equivalent of a specific processor architecture. We will discuss it and more in later chapters.

If you know basic c concepts, You could skip this chapter, But if you are not being in touch with c for a long time and needs a way to refresh the concepts, you could read this chapter, But i will try to keep it as simple as possible so that even beginners could start.

If you are interested, you should also master the c programming language along with the development of your own os. This is the technique i used to learn c. I learned c by making a small 2d game. Learning by doing and asking questions is a great way to practice. Asking question is not a bad thing , everyone who knows programming or any skill used to ask questions during their learning days.

Note that this chapter won't give you full coverage of c programming, but we will discuss everything necessary which can't be avoided when reading this book.

So lets dive into THE C PROGRAMMING LANGUAGE!!!!.

Hello , World

`Hello , World` is the first program used in most tutorials to introduce programming.
What this does is only print some text to the screen(Commonly "Hello , World" itself).

Lets see how we could print "Hello , World" to the terminal using c.

First, open `Notepad++` and type or copy paste the following code and save it:

```
1 #include <stdio.h>
2
3 int main(){
4       printf("Hello , World");
5 }
```

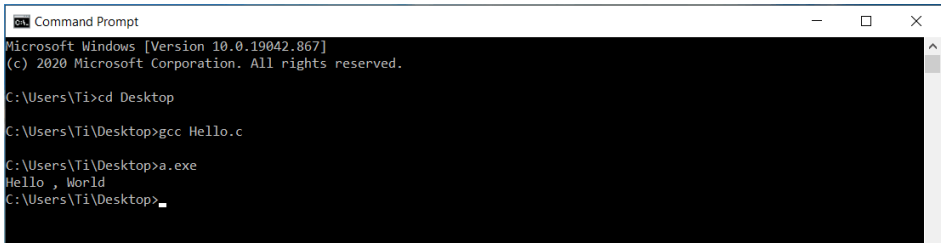
Please note that you need to give `.c` as file extension.

Compile the code by opening `cmd`, going to the directory where you saved the code using the `cd` command and finally compile it using this command:

```
gcc code.c
```

You need to replace `code.c` with the file name you saved.
Now you will see an executable file named `a.exe`
Now type `a.exe` in that command prompt itself to start it.

You will see something like this in your command prompt:



This is a simple program to display a message to the screen, You could replace content in the printf function inside quotes for example if you put that line like:

```
printf("Hello David");
```

It will print `Hello David` to screen.
Here the computer tries to print whatever is present in the " " Quotes.
`printf` is a function which is predefined in the `stdio.h` header file which we included at the very top. We will learn what a header file is later.

A thing to note is that after every statement in a line, we need to put a `;` at the end.

You could also see a portion:

```
int main(){

}
```

This is where the computer starts execution, So every program needs this main function to start execution.

Please note that you cannot call the printf function to display to the screen when developing the os as what this printf function does requires an operating system to display it. And as the only operating system in our virtual environment when testing the os is our os, We need to build some code so that when it is called, The text would be printed. We will see that later.

Lets now start learning the basics of c.

Data Types

`Data types` are a concept introduced to hold data. different data types have different storage capacities. Every data type have its own use.

Basic Data Types

Basic data types include `int` , `char` , `float` , `double` etc. These data types are generally used to hold arithmetic values. We can prefix `signed` and `unsigned` keywords before these keyword to alter its data holding limit.

We will learn about `int` and `char` data types.

int Data Type

`int` is the data type specifically used to store larger numbers. The size of int data type is mostly the bit length of the cpu it is running on.

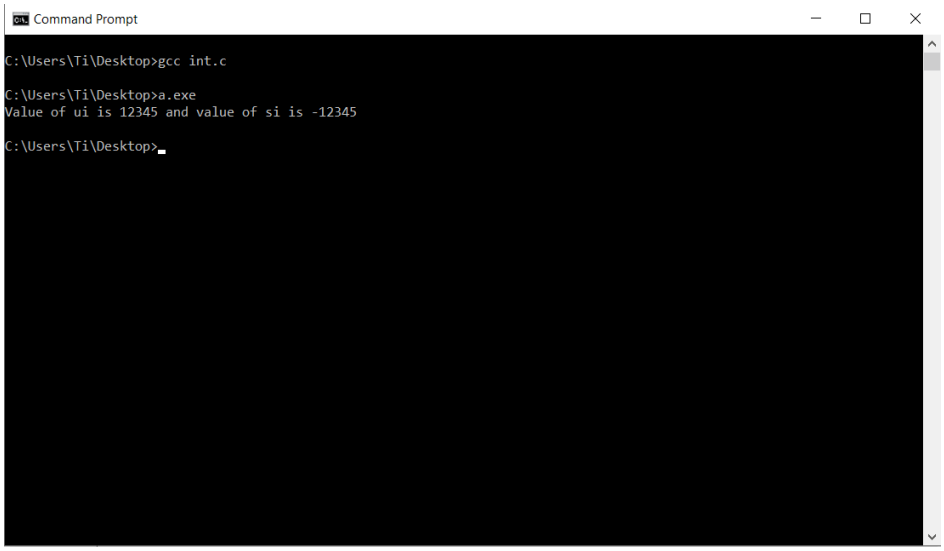
`unsigned int` can hold a value from `0` to `65,535` or `0` to `4,294,967,295`

`signed int` can hold a value from `-32,768` to `32,767` or `-2,147,483,648` to `2,147,483,64`. By default integers are signed so, You do not need to put signed keyword before int keyword to make it signed.

Lets see how we can assign and print some value from both `signed` and `unsigned int`.

```
1 #include <stdio.h>
2
3 int main(){
4
5      unsigned int ui = 12345;
6      signed int si = -12345;
7
8      printf("Value of ui is %d and value of si is %d\n" , ui , si);
9
10 }
```

compile and run the program, we will see something like this:



You can see that both the negative and positve numbers get printed. But the main effect of signed and unsigned keywords work during branching operations. We will see that later.

Here, We first assigned some values to the variables with the = operator. Then you can see the characters `%d` in `printf` function two times. When executing the program, computer will replace the first `%d` with the value of `ui` and the second `%d` with the value of `si`. `%d` helps us printing integer values which is passed after ‘,’ token.

The characters `\n` is put so that it puts a line break in the output.

Please try assigning values to both signed and unsigned int which crosses its limit and try printing it. Look what the result will be and try studying the reason.

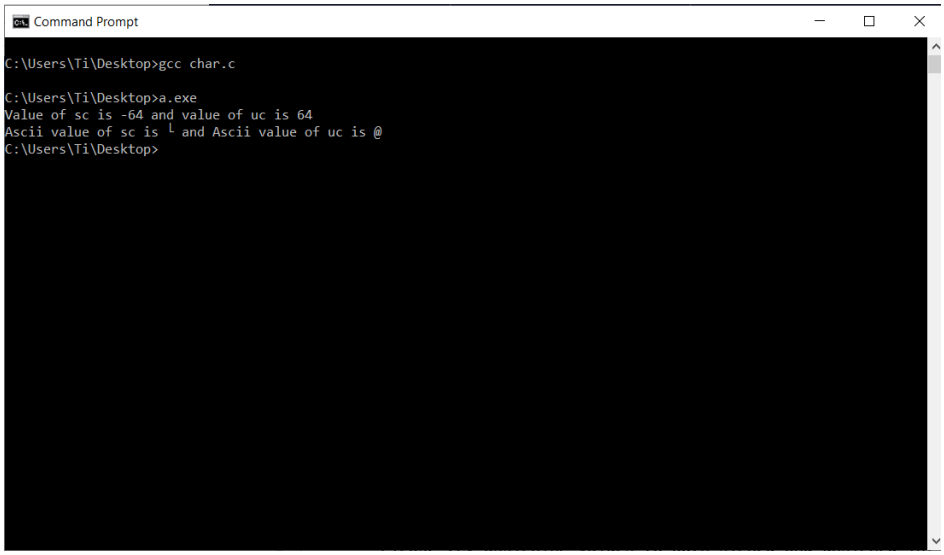
char Data type

`char` data type is specifially used to store `ascii` characters.
it is a `one byte` data type. `signed char` could store value from `-128` to `127`. `unsigned char` could store value from `0` to `255`

```
1 #include <stdio.h>
2
3 int main(){
4
```

```
5       signed char sc = -64;
6       unsigned char uc = 64;
7
8       printf("Value of sc is %d and value of uc is %d\n" , sc , uc);
9       printf("Ascii value of sc is %c and Ascii value of uc is %c" , sc , uc);
10
11  }
```

Output will be like this when compiled and executed



You could see in the source code that we used characters `%c` in it. `%c` is used to print the ascii character of value specified to it after the ‘,’ token

void Data Type

`void` is a data type used to represent `absense` of value. We will see the use of it later.

Derived Data Types

`Derived data types` are data types which are derived from fundamental data types like `int` , `char` etc…

Derived data types include:

1) **Pointers**
2) **Arrays**
3) **Structures**
4) **Unions**
5) **Functions**

Pointers

A `pointer` is a data type which holds `memory addresses`. This could be used to access data in a specific location. Every data in computers memory have an address, If we know the address of some specific data that we want, we could use pointers to access it.

Pointer data type do not stand on its own. We need some other keywords to work on pointers

We could first try declaring a `char pointer`. Have a look at the following code:

```c
#include <stdio.h>

int main(){

	char *Pointer = "Pointers are powerfull\n";
	printf("%s" , Pointer);

}
```

Have a look at the output:

We have declared a pointer by prefixing a `*` before a variable name. The one we declared is a character pointer. If you want to declare a char pointer with `addr` as pointer name, we could declare it as:

```
char *addr
```

To assign it with the address of a string such as `"Pointers are useful"`, Just declare it like this:

```
char *addr = "Pointers are useful";
```

You could see that we used the character `%s` in the code. `%s` accepts an address to a string. We passed the address by typing the name Pointer after ‘,’

We could now study pointers deeply. Look at the code up above. Here, what it does is, it declared a character pointer named addr and assigned it with the address of the string `"Pointers are useful"`. With that code we are only assigning the address of that string, not the string itself.

I will say why we used the keyword char to declare the pointer, Have a look at the following code:

```
#include <stdio.h>

int main(){

     char *Pointer = "COMPUTERS are powerfull\n";
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
     printf("%c" , *Pointer);

     Pointer++;
```

```
30      printf("%c" , *Pointer);
31
32 }
```

Lets see the output:

Here, we have printed the string `"COMPUTERS"` on screen. At the very top, we have declared a variable and assigned it with the address of a string. Then we have printed a character like this:

```
printf("%c" , *Pointer);
```

Here, we included a `*` before the variable name. What this does is to get whatever is present in the address specified in the variable Pointer.

If we didn't put the star before it, it will print the data contained in the variable Pointer which is an address, but as we put a star before it, it will take the data in the variable named Pointer as an address and returns whatever is located at that address.

By default the variable Pointer points to the first character of the string. So after printing the first character, We issued the command:

```
Pointer++;
```

What it does is only `increment` the address in Pointer by `one byte`(which is the `size of data type char` defined for the pointer). Now if we print the data pointed by the Pointer variable, We will print the second character of the string and this process continues with each increment and printing to the screen.

We could see how to get the address of a variable, for example the address of an integer and print whatever is in it:

```c
#include <stdio.h>

int main(){

	int num = 2021;
	int *pnum = &num;
	printf("address at pnum is %d and value at pnum is %d" , pnum , *pnum);

}
```

Output:

Here, we first declared an integer named `num` and assigned it with `2021`. Then we issued the following command:

```
int *pnum = &num;
```

This will declare an `integer pointer` named `pnum` and assign it with the address of value in `num` with the `&` symbol. The `&` symbol will return the address of value in variable `num`. Then we issued the following command:

```
printf("address at pnum is %d and value at pnum is %d" ,
pnum , *pnum);
```

First we printed whatever is present in the variable `pnum`(As we assigned it with the address of `num`, this will print the `memory address` of `num`). Then the command `*pnum` will take whatever is present in the variable `pnum` as an address and tries to print the data pointed by that address which in this case is the value `2021`.

Let's now see how we could assign values to a specific memory address using pointers:

```c
#include <stdio.h>

int main(){

    int iv = 0;

    int *address = &iv;

    *address = 3377;

    printf("%d\n" , iv);

}
```

Output

First, We created an integer named `iv` and assigned it with `0`. And in the second line, we created an integer pointer named `address` and assigned it with the address of variable `iv`.

The next line is as follows : `*address = 3377;`. Here what it does is that, it first takes the value in variable `address` as a memory address and assigns the value `3377` to that address.

Here, as we previously assigned the address of variable `iv` to the variable named `address` , The line : `*address = 3377;` will assign `3377` to the memory location of the variable `iv`.

We confirmed that the value got assigned by printing the value in `iv`.

So, This way we could use pointers to also **ASSIGN** values.

Arrays

`Arrays` are a collection of specific data type. An `integer array` means a collection of `integers` and `char array` means a collection of `characters`. `Arrays` are always `one after other`. We can access each element of array with its `index`. For example if we want to access the `first` element of an array we pass `0` as the index, and to get the `second` element, we pass `1` as the index and this process goes on…..

Lets see how we could declare and access an array of integers:

```c
#include <stdio.h>

int main(){

    int num[5];

    num[0] = 12;
    num[1] = 23;
    num[2] = 34;
    num[3] = 2020;
    num[4] = 8281;

    printf("%d %d %d %d %d" , num[0] , num[1] , num[2] , num[3] , num[4]);

}
```

Output:

Here, we declared an `integer array` of `5` elements with this code:

```c
int num[5];
```

We can access or assign all of the elements with `num[0]` to `num[4]`. Please note that the index of an array starts from `0` to `size_of_array - 1`. Here, it is from `0` to `4`.

Please look how we assigned values to each element and printed it.

Now let me reveal a secret. The arrays work somewhat as same as pointers. The only difference is that pointers use `*` and arrays use `[]`.

Let me show how we could access the array with pointers, Have a look at the code below:

```
#include <stdio.h>

int main(){

      int num[5];

      num[0] = 5;
      num[1] = 4;
      num[2] = 3;
      num[3] = 2;
      num[4] = 1;

      printf("%d %d %d %d %d" ,*(num + 0),*(num + 1),*(num + 2),*(num + 3),*(num + 4));

}
```

Output:

Here, We declared the array and assigned the value using the technique we learned just before this.

But have a look at how we printed it, It is different from what we have previously done. We've used the folowing technique:

```
*(num + 1)
```

What this does is like this : It first does thing which is inside the brackets, It increments `4` to the `num` variable(We have actually incremented `1`, but as the

size of datatype of the variabe num is 4 bytes , incrementing one will practically increments four bytes, incrementing two will increase eight bytes and so on), So now the num variable points to the second integer which is just after *(num + 0).

And as we have put a * before (num + 1), it will take (num + 1) as an address and take whatever is present in that address. This will practically access the second integer, likewise *(num + 2) will access the third integer.

I will request you to play with these concepts and do some more research so that you could learn more about this.

We will use pointers extensively when developing our operating system such as when we make the display driver. In 32 bit protected mode(We will explain that later), we need to put the data we want to print to the screen to a specifc address in memory, Which the video card will render to the screen, But for now, as i have requested , please work with pointers for some time to get the full idea of pointers

Lets now look at character array and work with pointers on it.

```
1 #include <stdio.h>
2
3 int main(){
4
5      char ca[5];
6
7      ca[0] = 'G';
8      ca[1] = 'A';
9      ca[2] = 'M';
10     ca[3] = 'E';
11     ca[4] = 'S';
12
13     printf("%c %c %c %c %c" ,*(ca + 0), *(ca + 1), *(ca + 2), *(ca + 3), *(ca
+ 4));
14
15 }
```

Output:

Here, the only difference is that, Instead of printing numbers, We just printed characters. You can try studying this code and move further.

I believe you are getting clear about the concept of pointers. You should further study pointers your own, there are many more concepts relating to pointers such as pointer to pointer etc….

Functions

Functions are a block of code which could be called and does a specific task. You have already used a function which is the main function:

```
int main(){

}
```

This is an example of a function and it is named `main`. We could create our own functions if we need. Look at the following example to get to know about functions:

```
#include <stdio.h>

int add(int a , int b);
int sub(int a , int b);

int main(){

      int num = add(150 , 50);
      printf("%d\n" , num);

      int num2 = sub(200 , 50);
      printf("%d" , num2);
}

int add(int a , int b){
      int c = a + b;
      return c;
}

```

```
20 int sub(int a , int b){
21      int c = a - b;
22      return c;
23 }
```

Output:

Let me explain it, Take a look at the `add` function that we have just created:

```
int add(int a , int b){

    int c = a + b;
    return c;

}
```

We have defined three integer variables here : `a` , `b` and `c`.
`a` and `b` are in the round brackets `()`. This means that the value of `a` and `b` will be given when calling it.

we called this function from the main function like this:

```
    int num = add(150 , 50);
```

Here, what the computer will do at runtime is that it passes the number `150` to the integer variable named `a` in `add` function and `50` to the integer variable named `b` in the `add` function.

After passing these arguments the computer starts executing the code in `add` function.
In that function, the code adds the values in `a` and `b` and stores it in `c`. Then it

returns the value in `c` to the variable `num` in main function. Then the main function prints it. We have also seen another function named `sub`.

```
int sub(int a , int b){

    int c = a - b;
    return c;

}
```

There is no special difference in the working of the code when this function is called with:

```
    int num2 = sub(200 , 50);
```

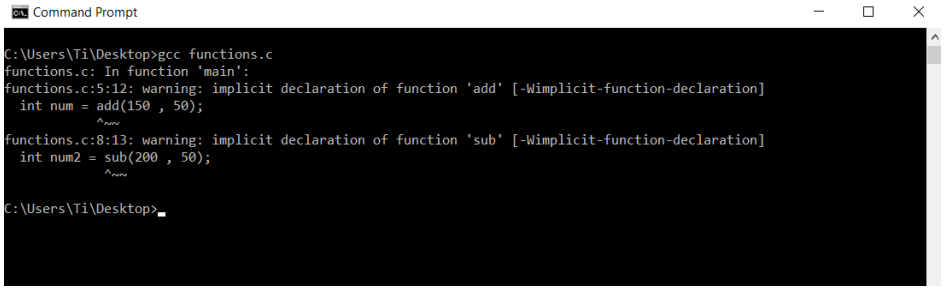
The only difference is that, the two numbers which got passed will go under a substraction operation and it returns the result to the `num2` variable in main function.

At the very top of the code you could see the following code:

```
    int add(int a , int b);
    int sub(int a , int b);
```

What this does is only telling the compiler that we will include code for those function later under the main function. If you do not do that before the main function, the compiler would stuck at what it is doing when it encounters the function call to `add` and `sub` functions because it haven't seen a function like that to jump to it.

You could try removing that two lines from the top and it will print some error message like this:



Lets learn something about the `return type` of a function. You can see that i typed the `add` function like this:

```
int add(int a , int b){

    int c = a + b;
    return c;

}
```

Not like this:

```
add(int a , int b){

    int c = a + b;
    return c;

}
```

The `int` keyword we gave before the function name `add` is used so that the compiler could get an idea about the data type of variable which that function will return using the `return` keyword. Here we have returned a variable named `c` which is of `int` type, This is the reason why we `prefix` the `int` keyword before the function name `add`.
The same rule applies also to the `sub` function.

You cannot define a certain data type as the return type and return variable of another data type. Like this:

```c
#include <stdio.h>

char add(int a , int b);

int main(){

     int num = add(150 , 50);
     printf("%d\n" , num);

}

char add(int a , int b){
     int c = a + b;
     return c;
}
```

Some compilers will show error when doing something like this but our compiler haven't shown any error , but you could see when executing that program that the output will be wrong

Output:

It is possible so that you can make a function which do not return any value, for that, you could use the `void` keyword.

```c
#include <stdio.h>

void printHello();

int main(){
     printHello();
}

void printHello(){

     printf("Hello!!");

}
```

Output:

Here, we have returned no value. We just jumped to a function and printed some characters to the screen from there. Then the control goes back to the main function and when at the end of the main function, the program itself terminates.

MORE INFO

You could return from any function(with or without return type) from anywhere in the code with the return keyword. You only need to type return; and if that code gets executed , the program control will go back to the function which called it.

You could make a function(with or without return type) With many , less or no arguments Like this:

```
void print();
int add();
void add(int a , int b);
int add (int a , int b);
```

You could call any type of function(with or without arguments) Like this:

```
PrintMessage();
Add(a , b);
```

Please note that the name of the function and/or variable is of your choice. I used these names just for example

I will suggest you to play with the concept of functions and also the overall concepts that we have covered this far before jumping to the next section.

Structures

Structures help us to combine a collection of data type into one packet. These data types may or may not be of the same type, Let's see an example:

```
#include <stdio.h>

struct car{
      int numberOfTyres;
      int price;
      char topSpeed;
};

int main(){
      struct car goldenCar;
      struct car yellowCar;

      goldenCar.numberOfTyres = 4;
      goldenCar.price = 2000;
      goldenCar.topSpeed = 100;

      yellowCar.numberOfTyres = 2;
      yellowCar.price = 4000;
      yellowCar.topSpeed = 110;

      printf("GOLDEN CAR : %d , %d , %d\n", goldenCar.numberOfTyres,
goldenCar.price, gol\
denCar.topSpeed);
      printf("YELLOOW CAR : %d , %d , %d\n", yellowCar.numberOfTyres,
yellowCar.price, ye\
llowCar.topSpeed);
}
```

Output

Here, We first created a structure named `car` with the `struct` keyword. Then we included three variables named `numberOfTyres` , `price` and

`topSpeed`. After closing the structure with `}`, we put a `;` denoting the end of that structure.

In the `main` function, we created two instances of the structure named `car` with it's name as `goldenCar` and `yellowCar`.

We did it with the command `struct car goldenCar;` and `struct car yellowCar;`

This command will allocate memory space to hold values in the structure named `car` and we could access those areas with its name (`goldenCar` and `yellowCar`).

Both `goldenCar` and `yellowCar` will have separate space to hold all of the three variables in the structure `car`.

Then we assigned values to variables in both of the instances of structure `car` by combining the instance name and variable name with a dot(.).

In the command : `goldenCar.numberOfTyres = 4;`, we assigned the value 4 to the variable `numberOfTyres` of instance `goldenCar` of `car` structure.

The next two lines assigned values to the rest of the variables in the instance of structure `car`.

Later, we assigned values to variables in the instance named `yellowCar` of the `car` structure.

At last we printed all of the variables in both `goldenCar` and `yellowCar` instance of `car` structure.

Branching

`Branching` is a concept which is used so that you could execute some piece of code based on some conditions. The keyword used for this is the `'if'` Keyword.

Lets look at an example:

```c
#include <stdio.h>

int main(){

	int i = 2;

	if(i == 2){
		printf("The number in i is 2");
	}

}
```

Output:

Here, We first assigned the value `2` to the integer variable named `i`. Then we checked in the `if` condition whether the value in `i` is `2` or not with the `==` operator.

As the value in the variable `i` is `2`, The program enters inside the condition and prints the message.

There are different operator which we could use in the `if` condition which are:

1) == : Equals to
2) != : Not equal to
3) >= : Greater than or equal to
4) <= : Less than or equal to
5) > : Greater than
6) < : Less than

You could use any of these operators in the if condition. Lets see another example:

```c
#include <stdio.h>

```

```c
int main(){

	int i = 2;

	if(i != 2){
		printf("The number in i is not 2");
	}

}
```

Output:

You could see that, no message is printed, This is because we initialized `i` with `2`, Then we checked in the condition if the value in `i` is not `2`. But actually as the value in it is `2`, The `printf` function will not execute and the program will exit doing nothing.

Lets now see how to make that program running by making only a small change:

```c
#include <stdio.h>

int main(){

	int i = 4;

	if(i != 2){
		printf("The number in i is not 2");
	}

}
```

Output:

Here, We initialized `i` with `4`. Then we checked if value in `i` is not `2` and as the value in `i` is not `2`, The `printf` function got executed.

We also have another keyword to work with branching which is the `else` keyword. Code in this `else part` will get executed if the condition checked with the `if` keyword is `not true`. Have a look at the following code:

```c
#include <stdio.h>

int main(){

	int i = 2;

	if(i != 2){
		printf("The number in i is not 2");
	}
	else{
		printf("The number is 2");
	}

}
```

Output:

Here we initialized `i` with `2`. Then we checked in the if condition if the value in `i` is not `2`. But as the value in `i` is `2`, The else part gets executed and the message got printed.

Now we will learn about `'else if'`. `'else if'` helps us check another condition if the condition we checked in the previous `if` keyword is false.

Have a look at the following code:

```c
#include <stdio.h>

int main(){

	int i = 2;

	if(i >= 3){
		printf("Value in i is greater than or equal to 3");
	}
	else if(i == 2){
		printf("Value in i is 2");
	}
}
```

Output:

At here, We first initialized `i` with `2`. Then we checked if the value in `i` is greater than or equal to `3`. But as the condition is false , the program will start evaluating the condition in `'else if'` part. Now as the `'else if'` condition is true, It will execute the code inside that `'else if'` part and will print the message.

MORE INFO

You could use the && operator to check whether two or more condition in the if keyword are true or not. Like this:

```
if(i >= 2 && i <= 5){

}
```

You could use || as an operator to check if any of two or more conditions specified are true or not. Like this:

```
if(i == 2 || i == 3){
```

```
}
```

In this case, the computer will start execution of code inside the if statement if any of the condition is true(If i is equal to 2 and/or i is equal to 3)

You could use nested if cases where one or more if cases are inside other if case Like this:

```
if(i == 2){

    if(y < 4){

        printf("Both conditions are true");

    }

}
```

We will now learn about the keyword `'switch'`:

`switch` is also a keyword which allows you to check conditions. This is used instead of the `'if'` keyword when we have to `check a lot of conditions`. Have a look at the following:

```c
#include <stdio.h>

int main(){

    int i = 3;

    switch(i){
            case 1:
                    printf("Value is 1");
                    break;
            case 2:
                    printf("Value is 2");
                    break;
            case 3:
                    printf("Value is 3");
                    break;
            case 4:
                    printf("Value is 4");
```

```
19                      break;
20              case 5:
21                      printf("Value is 5");
22                      break;
23      }
24
25 }
```

Output:

Here, We first declared a variable named `i` and initialized it with `3`.

The next line: `'switch(i)'` will say to compiler to check the value in variable named `i`.

Inside the switch statement after `{` , we put the `'case'` statement like this:

```
case 1:
```

This line will check if the value in `i` is `1` or not. If its `1` , then it will execute every code until a `break;` keyword.

In our case, it will only execute the following line:

```
printf("Value is 1");
```

But , in the program we developed , we initialized variable `i` with `3`. So the compiler will jump directly to the this line:

```
case 3:
```

And will execute every code inside this until it see a `break;`

And the only code between `'case 3:'` and `'break'` is `'printf("Value is 3");'` , the processor will only print the string `"Value is 3"`.

There is no limit that you could only call the `printf` function there. You could do any operation you want for eg : calling a function , doing some other branching or anything you want.

After the cpu run any of the case inside the switch statement and after the `break;` keyword, cpu will jump to the code after the switch statement(code after `'}'`).

Here, You could also try changing the value in `i` to get different results.

Please take the time to play with switch keyword.

You should now try playing with all of the concepts that we covered till here. This will help you attain more experience.

Looping

`Looping` is a concept which makes it practically possible to run same code repeatedly until a condition turns to be false.

To do that, we will use the keyword `'while'`. Have a look at the following code:

```
#include <stdio.h>

int main(){

	int i = 0;

	while(i <= 5){
		printf("*");
		i = i + 1;
	}

}
```

Output:

Here, We have defined a variable named `i` and initialized it with `0`. We then declared the while keyword and gave the following as condition:

```
i <= 5
```

This line checks whether the value in `i` is `less than` or `equal to 5`.

Computer will first check this condition when entering the loop and if its true, it will execute whatever is inside that loop. At the end of the loop at `}` , The computer will go to the top of that while loop and checks the condition and if it's true again, it will continue this work until the condition becomes false.

You could use every conditional operator that works in `if` keyword also at the while keyword. That conditional operators include == , `!=` , >= , <= , > and <.

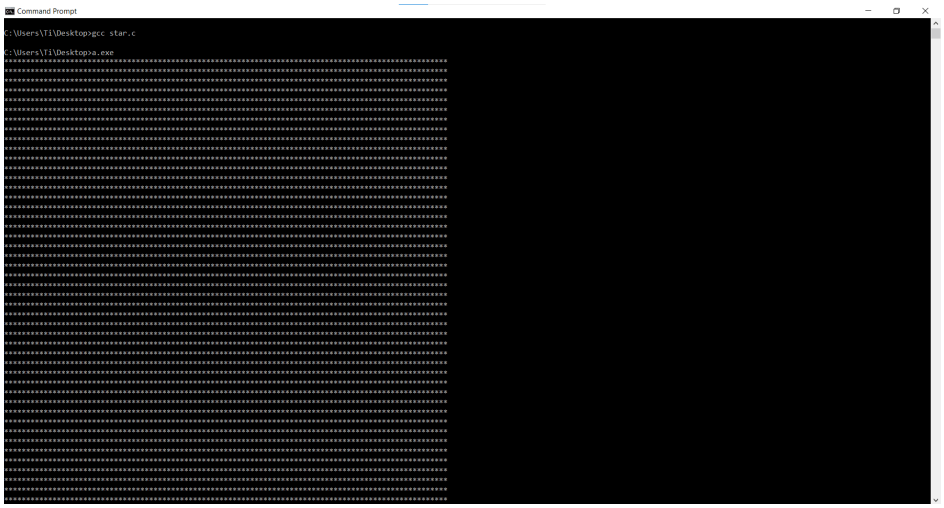
You could use the `&&` and `||` operators which we discussed for `'if'` keyword here.

You could also use nested loops which is a loop inside another loop : There is nothing preventing you to implement something like that. Lets see an example:

```
1  #include <stdio.h>
2
3  int main(){
4
5      int i = 0;
6      int y;
7      while(i <= 150){
8              y = 0;
9              while(y <= 100){
10                     printf("*");
11                     y++;
```

```
12              }
13              printf("\n");
14              i++;
15      }
16
17 }
```

Output:



Here `y++;` is same as `y = y + 1;` and `i++;` is same as `i = i + 1;`

Look at how fast your computer executes this program, This is the POWER and the USE of computers. You should now try to implement programs to make patterns like these. This is a very interesting job and it will help you a lot in this journey.

MORE INFO

You could use break; as a keyword so that when it executes, the cpu will exit from the loop it is currenlty running on

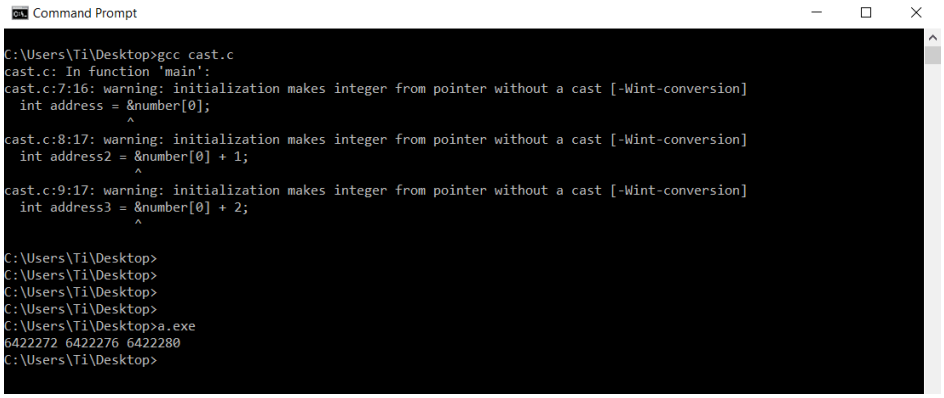
You could also use continue; as a keyword so that, when it executes, cpu will jump to the top of the current loop, which practically checks if the condition specified in that loop is true and if it's true, it will continue execution of code inside that loop

Type Casting

`Type casting` is a feature which allows us changing one data type to other. Also we could use this to change the behaviour of a special data type to the behaviour of another datatype. We do not have any special keyword to do this but we could do it. Have a look at the following code:

```c
#include <stdio.h>

int main(){

    int number[5];

    int address = &number[0];
    int address2 = &number[0] + 1;
    int address3 = &number[0] + 2;

    printf("%d %d %d" , address , address2 , address3);

}
```

Output:



Here you could see some warnings, You could forget about it for now. The output we got is the following `"6422272 6422276 6422280"`. You could see that each of this address is incremented with `4`. This means that address of `number[0]` is `6422272`,

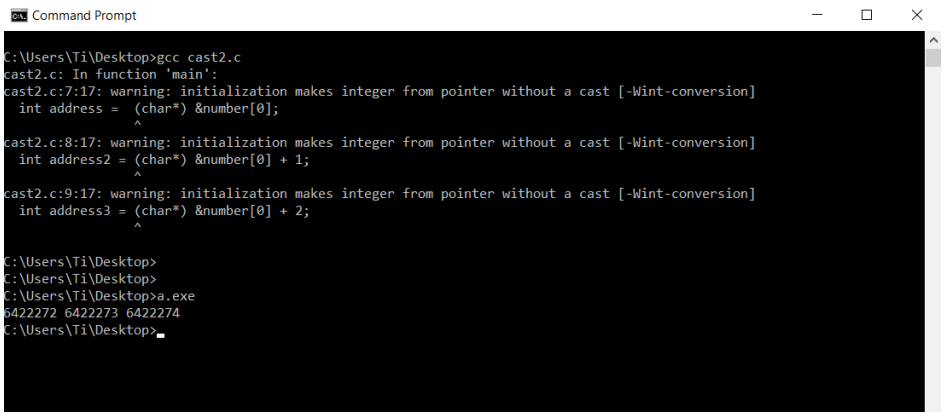
Address of `number[0] + 1`(or `number[1]`) is `6422276` and address of `number[0] + 2`(or `number[2]`) is `6422280`. This is a valid output as the size of an int is `4` bytes, incrementing it with `one` will always add `four`.

The `&` symbol here helps us get address of the variable.

But what if you want to increment the address by only `one byte` which helps you obtain each bytes of a number instead of getting the whole number. You could use casting like this:

```c
#include <stdio.h>

int main(){

    int number[5];

    int address =  (char*) &number[0];
    int address2 = (char*) &number[0] + 1;
    int address3 = (char*) &number[0] + 2;

    printf("%d %d %d" , address , address2 , address3);

}
```

Output:



Here, You could see in the output that each address is incremented by only `one` , not `4`.

What we does special in this code is that we added `(char*)`. This says to the compiler that we should only increment the value with the size of `char`(which is `one` byte), and this will result so that only `1` is added to the address for each increment.

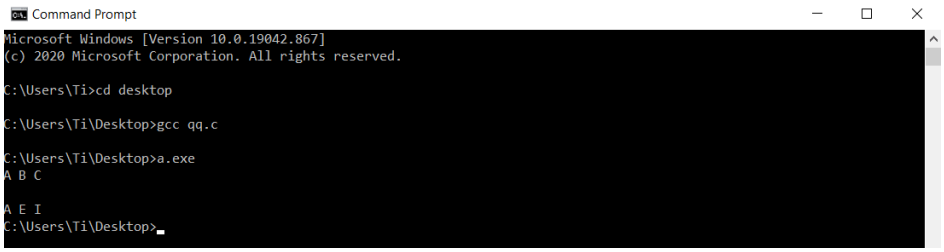
We added `*` to the `(char)` so that it tells the compiler that what we do is an operation on memory address so it looks like this now: `(char*)`.

If you did not understand this, Please try reading again, This is not a big deal.

I will show another example:

```c
#include <stdio.h>

int main(){

    char *string = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";

    char a = *(string + 0);
    char b = *(string + 1);
    char c = *(string + 2);

    printf("%c %c %c\n\n" , a , b , c);

    char x =  *((int*) string + 0);
    char y =  *((int*) string + 1);
    char z =  *((int*) string + 2);

    printf("%c %c %c" , x , y , z);
}
```

Output:



Lets study the code now:

The first line is as follows : `char *string = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";`

This line will declare a char pointer named `string` and assign it with the address of : `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"` in memory.

The next three lines are as follows :

```
char a = *(string + 0);
char b = *(string + 1);
char c = *(string + 2);
```

Here the line: `char a = *(string + 0);` will add `0` to the address specified in the variable `string`. The `*` before the variable named `string` will result in returning the data pointed by the address in `string` variable to the character variable `a`.

This assigns the first character pointed by the variable `string` to the character `a`;

The only difference in the second line is that instead of adding `0` to the `string` variable , we added `1` to it. This will result in returning the second character pointed by `string` variable to the character variable `b`.

The same follows also in the third line , adding two will result in returning the third character to the variable named `c`.

In three of the cases , You saw that incrementing the value in string variable by `one` result in returning the consecutive characters.

Then we printed the values in `a` , `b` and `c` with: printf("%c %c %c\n\n" , a , b , c);
This prints the consecutive characters `"A B C"` to the terminal.

Then we does another approach like this:

```
char x = *((int*) string + 0);
char y = *((int*) string + 1);
char z = *((int*) string + 2);
```

Here we applied an `integer casting` with `(int*)`.

The first line result in returning the character `'A'` To the variable `x`.
Then , in the second line we added `1` to the variable `string`.

Here as we casted it with `(int*)` , adding `1` will actually result in adding `4` which is the size of integer(`int`). And the outer most `*` in `*((int*) string + 1)` will obtain the character `'E'` pointed by address in `string + 1` and return it to the variable `y`.

The same applies to the third line and it obtains the character `'I'`. As we added `2` to the variable `string`, it will return the character at `2 * size of int` which is `2 * 4` which is `8`.

There are more details we could add to the topic `Type casting`, But for this tutorial, This much is enough.

NOTE

`It is a good idea to learn more on this topic which will help you in advanced implementations`

Arithmetic Operators

We have already covered one of the Arithmetic Operations which is the addition operation. We used this to implement examples programs. Here you could learn more:

```
#include <stdio.h>

int main(){

    int base = 4;

    int a = base + 2;
    int b = base - 2;
    int c = base * 2;
    int d = base / 2;

    printf("%d %d %d %d %d" , base , a , b , c , d);


}
```

Output:

The code and the output says it all, But let me explain it.

Addition could be done as:

```
a = x + y;
```

Substraction could be done as:

```
a = x - y;
```

Multplication could be done as:

```
a = x * y;
```

Division could be done as:

```
a = x / y;
```

Now, Have a look at the following code:

```c
#include <stdio.h>

int main(){

      int a = 2;
      int b = 4;
      int c = 6;
      int d = 8;

      a += 2;
      b -= 2;
      c *= 2;
      d /= 2;

      printf("%d %d %d %d" , a , b , c , d);


}
```

Output:

I assume that you have already figured out what this code does, But let me explain it:

Here, `a += 2` is same as `a = a + 2`

`b -= 2` is same as `b = b - 2`

`c *= 2` is same as `c = c * 2`

And finally `d /= 2` is same as `d = d / 2`

We finally have one more operator, Which is the `modulus` operator. `Modulus` operator returns a `remainder after integer division`. Have a look at the following code:

```
1  #include <stdio.h>
2
3  int main(){
4
5      int a = 12;
6
7      int b = a % 10;
8      a %= 10;
9
10     printf("%d %d" , b , a);
11
12 }
```

Output:

Here, `int b = a % 10;` first divides the value in `a` with `10` and returns the remainder to `b`. `12 / 10` returns `2` as remainder. so value in `b` will be `2`.

`a %= 10;` is same as `a = a % 10`, This will return `2` as remainder and it will be assigned to `a`.

Increment , Decrement Operators

Look at the following code:

```c
#include <stdio.h>

int main(){

    int a = 4;
    int b = 8;

    a++;
    b--;

    printf("%d %d" , a , b);

}
```

Output:

Here, `a++` is the increment operation and it is same as `a = a + 1` and `a += 1`.
`b--` is the decrement operation and it is same as `b = b - 1` and `b -= 1`.

You can try comparing this with the output.

Bitwise Operators

Bitwise AND and Bitwise OR Operators

Both the AND and OR operators do pure binary operations. Let's first learn Bitwise AND operation.

Take the binary value 11010 and 11011 as two variables. Doing an AND operation on these values looks like this:

```
11010
11011
_____

11010
```

Doing AND operation on 11010 and 11011 gives us 11010 as result.

In an AND operation, The cpu takes each bit in both of the variables and generates a corresponding binary value 1 if both of the bits are 1 and generates 0 if both or any of the bit is 0.

The Bitwise OR operation looks like this:

```
11010
11011
_____

11011
```

Here, The cpu takes each bit in both of the variables and generates a corresponding binary value 1 if any of the bit is 1 and generates 0 if both of the bits are 0.

In c, The symbol & could be used to perform AND operation and the symbol | could be used to perform OR operation.

Lets See an example:

```c
#include <stdio.h>

int main(){

    char aa = 3;
    char ab = 5;

    char AND = aa & ab;
    char OR  = aa | ab;
```

```
10
11      printf("%d %d" , AND , OR);
12
13 }
```

Output

Left shift and Right shift Operators

Both `left shift` and `right shift` operators shifts bits in a binary value to the specified side, the specified number of times.

Take binary value `00010` for example. Left shifting it by `one` will change that value to `00100`, Right shifting the value `00010` it by one will change it to `00001`.

You could shift the bits to any number of position, But as an example here , we will shift by `one`.

In c, The symbol << could be used as left shift operator and >> could be used as right shift operator.

Lets look at an example:

```
1  #include <stdio.h>
2
3  int main(){
4
5       char left = 3;
6       char right = 3;
7
8       left = left << 1;
9       right = right >> 1;
10
11      printf("%d %d" , left , right);
12
13 }
```

Output

Macros

`Macros` or `pre-processors` are a feature provide by the c compiler to do things at the compile time.

From the start of this book, we read about `Data types` , `branching` , `looping` etc. These things will take effect only when we run the executable.

`Macros` provide us the ability to do things at the compile time(when we compile the code with commands like : `gcc code.c`).

We will learn about two Macros here which are the `#include` and `#define` directives.

#include

The `#include` directive is used to include other c source files into the program we develop so that we could call function in that file and more.

All of the examples we have covered this far have `#include <stdio.h>` in the very top.

This will include the file named `stdio.h` into our source file. The `<>` brackets tells to the compiler that to look for the `stdio.h` file in the `c standard library folder`.

When developing our operating system, it is advised not to include any of these standard libraries as if any of the functions we call in this files contains

platform specific calls(Specifically calls to the operating system), It will result in run time errors.

But we can include our own files into the source files we create. Imagine if you want to create a file named `expl.h` and call a function we made in that file. To do this , create a file with that name and put the following code into it so that we could call it:

```
void out(){
	printf("Hello");
}
```

Now create a file named `func.c`(You can name it anything you want) and put the following code:

```
#include <stdio.h>
#include "expl.h"

int main(){
	out();
}
```

Now compile and run `func.c` file to get the output:

You can see that the message got printed. We called a function named `out`, in `expl.h` file to print something to the screen. This way you could include any file you need.

For that we used the `#include` directive with filename in `""` at the top of the file.

As what we does is printing to the screen, don't forget to include the `stdio.h` file

Also note that two of this files should be in the same folder to get compiled.

Now, Look at the following case:

The main file `func.c` is in a folder named `src` and you want to include the file `expl.h` which is contained in a sub folder named `header`(which is also inside the src folder), To do that, include the following line in the `func.c` file.

```
#include "header/expl.h"
```

You should now go to the `src` folder using command prompt using `cd` command and compile the `func.c` file

Please note that you should put `.h` as extension for the files included by the `#include` directive.

#define

The best way to explain `#define` directive is to show an example, Have a look at the following:

```
1 #include <stdio.h>
2
3 #define Message "WE ALMOST COMPLETED OUR C PROGRAMMMING TUTORIAL"
4
5 int main(){
6       printf(Message);
7 }
```

Output:

Here, what the line:

```
#define Message "WE ALMOST COMPLETED OUR C PROGRAMMMING TUTORIAL"
```

Does is only make the compiler to copy paste where ever it see the word `'Message'` with the string `"WE ALMOST COMPLETED OUR C PROGRAMMMING TUTORIAL"`

In the `printf` function We passed `'Message'` inside `()`. What this does is only replace the string `'Message'` with the string `"WE ALMOST COMPLETED OUR C PROGRAMMMING TUTORIAL"` at the compile time.

So it will look like this when compiling(But we can't see this):

```
printf("WE ALMOST COMPLETED OUR C PROGRAMMMING TUTORIAL");
```

Hexadecimal Notations

`Hexadecimal` is just a number system which allows us with ease of work with computers.
We Humans generally use `decimal` as the number system which have a base of `10`.
Base of `10` means we have a total of `10` number ranging from `0` to `9`.

The base `10` of `decimal` number system is said to be derived from the fact that humans have a total of 10 fingers. But anyway, when we work with computers, the efficient way is to use hexadecimal number system.

This system have a base of `16` which comprises of values from `0 - 9` and `a - f`

Hexadecimal value of `1` to `9` is same as that of `decimal`.
For eg: Hex of decimal value `1` is `1`,
Hex of decimal value `9` is `9`.

But the hex value of `10` is `a`.
11 = b
12 = c

13 = d
14 = e
15 = f

Now the hex of decimal value `16` is `10`

17 = 11
18 = 12
19 = 13
20 = 14
21 = 15
22 = 16
23 = 17
24 = 18
25 = 19
26 = 1a
27 = 1b
28 = 1c
29 = 1d
30 = 1e
31 = 1f
32 = 20

And this process goes on.
Hex of `255` is `ff`.
`255` or `ff` is the highest value that a `byte` could have.

A `byte` means `8 bits`. so in the value `ff` , The first `f` is took from the first `four bits` of the `byte` and the next `f` is took from the remaining `four bits`

So, The main advantage of the `hexadecimal` format allows us to get a `single hex digit` from `four bits`.

Each group of `4 bits` in `binary` is a `single digit` in the `hexadecimal` system. This makes it really easy to convert `binary` to `hexadecimal` numbers.

We could use `hexadecimal` number system in `c` by prefixing `0x` on a hex value, for eg: if we define it as `0xa`, this will get converted to the decimal value `10`.
`0xff` will be converted to decimal value `255`.

`Two hexadecimal characters` make `one byte`, `Four` of them make `2 bytes`, `six` of them make `3 bytes`. eg: `0xff` is `one byte`, `0xffaa` is `two bytes` and `0xffaa11` is `three bytes`.

Have a look at the following code:

```c
#include <stdio.h>

int main(){

    int a = 0x4;
    int b = 0xa;
    int c = 0xff;
    int d = 0xff12;

    printf("%d %d %d %d" , a , b , c  , d);
}
```

Output:

You could now do further studies relating to the hexadecimal number system if you want.

Comments

`Commenting` is a feature provided by almost every language which let us include documentation or other helpfull information along with the code.

Comments would be skipped at compile time.

We have two ways to comment inside our code in `c`. We could comment on a single line with `//` and comment on multiple lines with `/*` and `*/` , Have a look at the following:

```c
#include <stdio.h>

int main(){

    /*
            This is an example of multi line comment.
            This allows us include multiple lines.
            The main aim of comments is to document working of the code
    */

    printf("Hello"); // This is an example of single line comment

}
```

When compiled and executed, We won't have any errors, and the code would work fine.

Output:

Let's Have A Game

This section will help you to put what you have learned this far and implement it.

This is the challenge for you:

You have a variable named `i` , You can assign it with any value , but first assign it with `5` and try to print one star on the first line , two stars in the second line and , three stars on the third line and continue until the number of lines reaches the value of `i`.

If you assign `i` with `5`, This will be the output:

Now, Try to find a solution to this problem.

Solution

```c
#include <stdio.h>

int main(){

	int i = 5;

	int y = 1;
	int z;

	while(y <= i){
		z = 0;
		while(z < y){
			printf("*");
			z++;
		}
		printf("\n");
		y++;
	}
}
```

Programming in Assembly Language

Introduction

Every computers have a common way of working which follows: Input –> Processing –> Output. Processing is the powerfull and resource intensive area.

The Hardware which processes instruction in normal Desktop or Laptop computers is commonly called as a Micro Processor. Micro processors does three thing in common which are : Fetch –> Decode –> Execute.

The processor first takes an instruction from the memory, Then it will decodes the instruction which makes the internal circuitry ready for the execution of that single instruction and finally, it will do the operation specified by the fetched instruction.

I am not willing to make you mad by explaining theories, But i will explain only the necessary things which you can't skip.

What is an Assembly Language

Keeping it simple, Assembly language is a low level language which helps us teach computers do a task by explaining it in a step by step manner.

An assembler is a software which converts the program's we wrote in assembly language to pure binary which the processor could directly execute.

We have special instructions in assembly language to do the task we need. The addition operation will look something like this in assembly:

```
add x , 5
```

This instruction will simply add 5 to `x`. Copying values will look something like this:

```
mov numb , x
```

This will simply copy the value in `x` to `numb`.

Now, Lets see what the main role of an assembler is:

The processor won't be able to do things that we write in english or other human readable language.

Take the addition instruction

```
add x , 5
```

as an example: The assembler will convert this human readable code to binary. For this example, Please take the following hex values into consideration(Please don't take it as exact , this is only used to convey the idea):

```
0x83 0x45ff 0x5
```

The assembler will convert the add instruction to the hex value `0x83`, Then to represent the address of `x`, it converts it to the hex value `0x45ff`(Please note that this is not the exact value everytime, it will depend on the address of variable in focus). Now, to denote the addition of value 5, it will be converted to `0x5`

During the execution, The processor will first take the byte `0x83`.
It is predefined in the processor circuitry so that when it see the hex value `0x83`, to prepare the processor to fetch some of the next binary values to perform the addition operation.

Now, The processor fetches these values `0x45ff 0x5`

Finally, The processor adds `0x5`(5 in decimal) to the value located at address `0x45ff`.

Likewise, the processor have a table of binary values like these which represent each operation.

These values are called machine code.

There are different micro processor architectures in the market. Each processor architecture will have its own unique table of values(machine code). This means that programs written for a special processor architecture won't work in another processor architecture.

The Processor architecture we are going to work with is intel's x86 architecture.
This is the mostly used architecture currently and it is where Windows and Linux Mainly run on. Even though x86 is developed by intel, Companies like AMD also manufactures the same.

So at the conclusion, Assembly language programs are a processor architecture specific, low-level, Human readable routine which would be translated to a non-human-readable(binary) form also known as machine language, by a software commonly named as assembler.

What is a Compiler actually?

The main purpose of a compiler is not to convert the source code to binary.

We have discussed about the `add` and `mov` commands in x86 assembly language. There are many instructions that we can use to make our program in assembly.

To know what a compiler is, Please save the following c code to a file, name it `code.c` :

```
1 #include <stdio.h>
2
3 int main(){
4       printf("Hello , World");
5 }
```

Now, Go to where you saved the file with command prompt and type the following command

```
gcc -S code.c
```

You will now get a new file in the same directory. In my system, I got a file named `code.s`

Have a look at its content:

```
1      .file   "code.c"
2      .def    ___main;        .scl    2;      .type   32;     .endef
3      .section .rdata,"dr"
4  LC0:
5      .ascii "Hello , World\0"
6      .text
7      .globl  _main
8      .def    _main;  .scl    2;      .type   32;     .endef
9  _main:
10 LFB10:
11     .cfi_startproc
12     pushl   %ebp
13     .cfi_def_cfa_offset 8
14     .cfi_offset 5, -8
15     movl    %esp, %ebp
16     .cfi_def_cfa_register 5
17     andl    $-16, %esp
18     subl    $16, %esp
19     call    ___main
20     movl    $LC0, (%esp)
21     call    _printf
22     movl    $0, %eax
23     leave
24     .cfi_restore 5
25     .cfi_def_cfa 4, 4
26     ret
27     .cfi_endproc
28 LFE10:
29     .ident  "GCC: (MinGW.org GCC-6.3.0-1) 6.3.0"
30     .def    _printf;        .scl    2;      .type   32;     .endef
```

This is an assembly code equivalent to the c code we wrote. So this is the point, a compiler is only a program which translates the code we write in a higher level language like c, to a lower level language like assembly. Then the c compiler will call its own assembler to assemble the assembly code which the compiler generated.

You don’t need to worry about the complexity of assembly code you saw here. We will not write assembly programs like this. We will only use a neat style to learn this. And mainly we are not going to use the assembler which

comes with the compiler. This is why we have downloaded nasm as our assembler.

You just saw the amount of lines created just for our hello world program. This is why no one these days write code in assembly and it is also the reason behind the development of languages like c.

Assembly language is processor specific, and if we write code in assembly for a specific processor, we wont be able to make it work on other processor architectures. If you want to do that, You may need to write that program in assembly language of the processor you want to port to from scratch. This is where a compiler come. If we write our os in c or similar language, we will be able to port the program to other processor architectures if the processor vendor provide a c compiler for their processor. But we wont be able to port assembly programs like this.

This is why we write our os in c, But we surely need to write some part of the code in assembly. When porting your os to other processor architecture, You may need to change every assembly code you wrote for your os to the equivalent code for the processor architecture you want to port to.

x86 Processor data sizes

The x86 processor defines different data sizes with its name and its size. Here is the list of it:

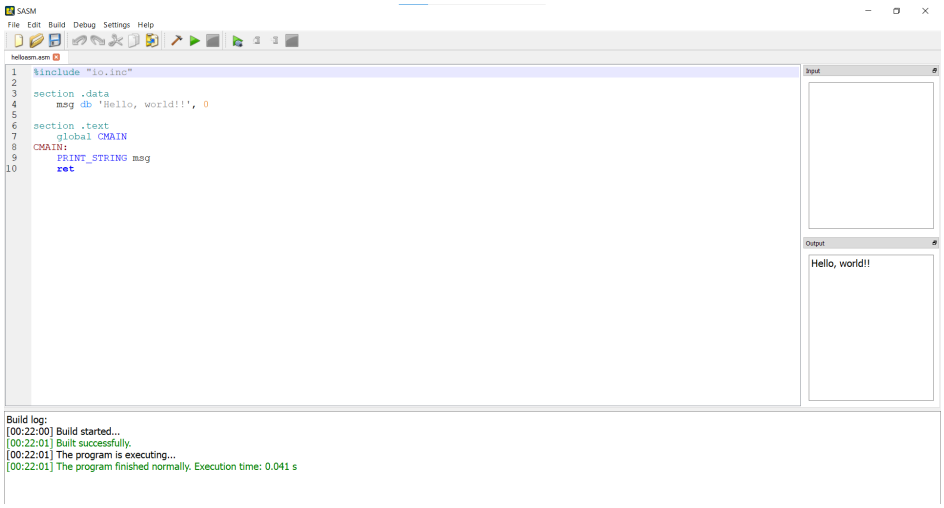
Word: a 2-byte data item
Doubleword: a 4-byte (32 bit) data item
Quadword: an 8-byte (64 bit) data item
Paragraph: a 16-byte (128 bit) area
Kilobyte: 1024 bytes
Megabyte: 1,048,576 bytes

Assembly Hello , World

We are now going to use the software named sasm that we have downloaded. First open it and create a new project and include the following code into it:

```
%include "io.inc"

section .data
    msg db 'Hello, world!!', 0

section .text
    global CMAIN
CMAIN:
    PRINT_STRING msg
    ret
```

Run it by pressing the play button and we will get the following output:



We will explain the working later. For now, We could jump to the next section.

Registers

Now, we will learn some theories.

The main function of a processor is to process data. These data's are mainly stored in main memory(also known as RAM or Ramdom Access Memory). As the job of a processor is to process data, It may need to access memory often.

Here, We have to think about a factor, which is speed. The processor is very much faster than RAM and accessing it frequently will decrease it's

processing speed to a greater extend. It is also an inefficient way as accessing it frequently will put a lot of traffic on the system bus.

This is where the concept of register's come. Registers are the memory inside the processor. The technology used to implement registers are so advanced so that the processor could execute at a maximum speed.

But we can't use the technology used to implement registers to build a main memory. If this is practical, We could think of an architecture with no registers, With only a main memory. But this idea will only be in theories. The registers in the processor these days are either 32 bit or 64 bit in size. But when talking about main memory, We will atleast need 2 or 4 GB.

If we use the technology used to implement registers to build main memory, It will cost you as much as money which only a few could handle. This is the reason why we implement computers with RAM as a slow memory –> cache as a faster memory that ram –> And finally registers as an even faster memory than cache.

The following is the list of devices in computer with increasing amount of speed and decreasing amount of capacity:

1)**Hard disks** – Permanent storage
2)**Ram** – Data goes off when the computer is off
3)**Cache** – Data goes off when the computer is off
4)**Registers** – Data goes off when the computer is off

The registers are grouped into three categories:

1)**General registers**
2)**Control registers**
3)**Segment registers**

The general registers are further divided into the following groups:

1)**Data registers**
2)**Pointer registers**
3)**Index registers**

General Registers

Data Registers

There are four data registers used for arithmetic and other operations. These are the 64 bit data registers:

1)**RAX**
2)**RBX**
3)**RCX**
4)**RDX**

Lower halves of the 64 bit data registers can be used as 32 bit data registers which are:

1)**EAX**
2)**EBX**
3)**ECX**
4)**EDX**

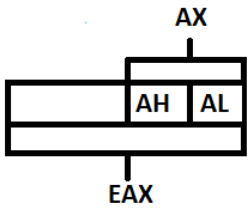
Lower halves of the 32 bit data registers can be used as 16 bit data registers which are:

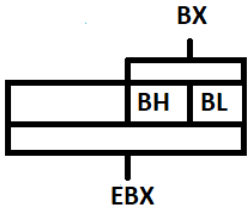
1)**AX**
2)**BX**
3)**CX**
4)**DX**

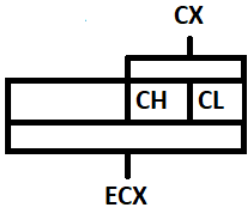
Higher and Lower halves of 16 bit data registers can be used as 8 bit data registers which are:

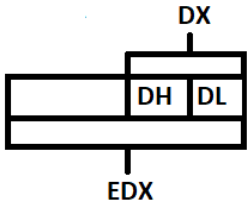
1)**AH** , **AL**
2)**BH** , **BL**
3)**CH** , **CL**
4)**DH** , **DL**

Have a look at the following images which shows division of 32 bit data registers:









The **AX** is the primary accumulator: It is used in input/output and most arithmetic operations.
The **BX** is known as base register: It could be used in indexed addressing(We will see this later).
The **CX** is known as the count register: It is mainly used to store loop count.

The **DX** is known as data register: It is also used in input/output operations. It is also used along with the **AX** register for arithmetic operations.

Pointer Registers

There are three pointer registers. 64 bit ones are named:

1)**RIP**
2)**RSP**
3)**RBP**

32 bit ones are named:

1)**EIP**
2)**ESP**
3)**EBP**

16 bit ones are named:

1)**IP**
2)**SP**
3)**BP**

Instruction Pointer (IP) - This register stores the address of next instruction to be executed. x86 architecture use segmented addressing. `IP` is associated with the `CS` register. `CS:IP` gives the complete address of the next instruction to be executed(We will see this later).

Stack Pointer (SP) - Mainly used to point to the top of stack. `SS:SP` Gives the complete address in segmented addressing(We will see this later).

Base Pointer (BP) - This is mainly used to access memory relatively. BP - A special offset gives access to variables in memory. `SS:BP` gives the complete address in segmented addressing(We will see this later).

Index Registers

There are 2 index registers. 64 bit ones are named:

1)**RSI**
2)**RDI**

32 bit ones are named:

1)**ESI**
2)**EDI**

16 bit ones are named:

1)**SI**
2)**DI**

Source Index (SI) - This is mainly used as source index for string operations.

Destination Index (DI) - This is mainly used as destination index for string operations.

Control Registers

Many instructions in x86 architecture involves comparisions and mathematical operations. This will change the status of some flags and some other conditional instructions test the values of these flags to influence the control flow.

The common flag bits include:

1)**Overflow Flag (OF)**
2)**Direction Flag (DF)**
3)**Interrupt Flag (IF)**
4)**Trap Flag (TF)**
5)**Sign Flag (SF)**
5)**Zero Flag (ZF)**
6)**Auxiliary Carry Flag (AF)**
7)**Parity Flag (PF)**
8)**Carry Flag (CF)**

I am not going to explain and overwhelm you with lot of theories about this. If you need more information about this, You could google about x86 control registers.
Not learning this deeply wont affect you for starting in this field. But when you are planning to go advanced, It's a good idea to learn about this.

Segment Registers

Segments are specific areas in a program for containing Data , Code and Stack. x86 initially was a 16 bit architecture. So we could only access data in memory with 16 bit address. This will limit the size of ram that could be implemented. Segmented memory is used to allow us access more memory. Segment registers are the solution used to solve this problem.

The Registers include:

1)**Code Segment (CS)**: Used to point to executable code. This register stores the starting address of the code segment.

2)**Data Segment (DS)**: Used to point to Data. This stores the starting address of Data Segment.

3)**Stack Segment (SS)**: Used to point to the starting of the stack segment. Stack segment is where variables which are local to functions are stored.

x86 Processor Endianess

Processor Endianess defines how a `multi-byte` data type is stored in memory.
The early computers implemented a `4 bit` or `8 bit` processor architecture.

x86 architecture was a 16 bit architecture and data items in it's registers qualify to be named as `multi-byte` data type.

The two types of endianess which exists are named as `Little Endian` and `Big Endian`. x86 uses `Little Endian` architecture. `Little Endian` architecture stores multi byte data in a reverse order in memory.

By reversing, it doesn't mean the first `bit` is stored last and last `bit` is stored first. The reversing get affected at the `byte` level which means that first byte of a multi-byte data item get stored last, Second byte get stored second last and goes until the last byte gets stored first.

When copying the data from memory to any register, another byte reversing will be done which practically makes the number in the real form.

For Eg: Consider `0xaabb` as a multi-byte data. In memory, It will be stored as `0xbbaa`. After copying it to any register, It will be back in normal form as `0xaabb`.

Consider the following `32 bit` data :

```
0xaabbccdd
```

This will be in memory as `0xddccbbaa`.

It will be in normal form like this:

```
0xaabbccdd
```

When in register.

When talking about the `C` Programming language in x86 architecture, The keyword `char` won't be affected by Endianess as `char` is a single byte data type, But the `int` keyword will always be affected by Endianess as `int` keyword always uses 4 - 8 bytes.

The other architecture which exists is the `Big Endian` architecture. This architecture uses the normal way of storing which stores the first byte first and continues untill the last byte get stored last.

Commands For Register Operations

Most of the operations we do invloves registers. So, We have different commands to work with registers. We will discuss about the important ones.

mov Command

The mov command is used to copy values from one location to other. Following is the basic syntax:

```
mov destination , source
```

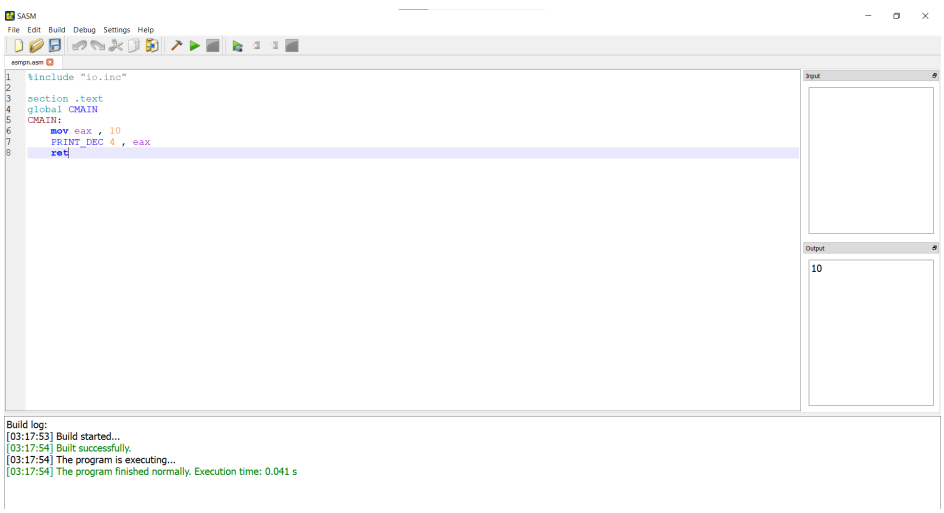
Consider a situation where you want to copy the value 10 to the eax register. The code will look like this:

```
mov eax , 10
```

Here's how we could print it in sasm.

```
%include "io.inc"

section .text
global CMAIN
CMAIN:
    mov eax , 10
    PRINT_DEC 4 , eax
    ret
```

Output



You only need to look at the code after `CMAIN:`
The command `mov eax , 10` only copies the value 10 to the eax register.

In `PRINT_DEC 4 , eax`, `PRINT_DEC` is not a command. It is only a routine defined in the `io.inc` file included at the top of the code. Here, It helps us convert binary value in `eax` register to decimal form and print it as an output.

`4` in `PRINT_DEC 4 , eax` tells the routine that the operation is done in a four byte data item(Here it is the `eax` register).

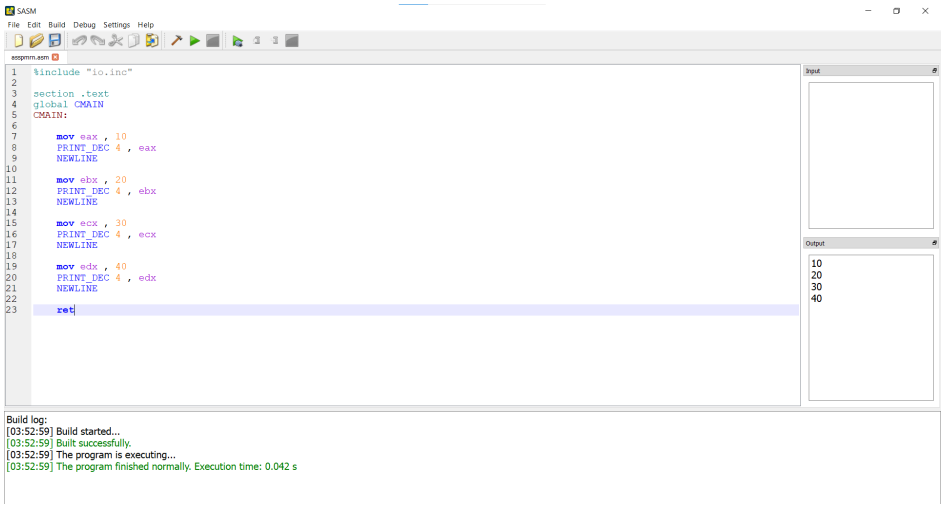
You do not need to focus on this printing function as we wont have this routine when developing our os. We will build our own routine for that.

The very last command `ret` will practically terminate that program, The function of `ret` command is not to terminate a process. We will discuss that later.

We could do this operation in more register, Look at the following:

```
1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax , 10
8      PRINT_DEC 4 , eax
9      NEWLINE
10
11     mov ebx , 20
12     PRINT_DEC 4 , ebx
13     NEWLINE
14
15     mov ecx , 30
16     PRINT_DEC 4 , ecx
17     NEWLINE
18
19     mov edx , 40
20     PRINT_DEC 4 , edx
21     NEWLINE
22
23     ret
```

Output



Here, We tested the `mov` command with different registers and it worked! The line `NEWLINE` will put a line break in the output. It is not a command in x86 architecture. `NEWLINE` is pre defined in the `io.inc` file we included at the top.

add Command

As the name say's, add command is a command to perform the addition operation.

The syntax is as follows:

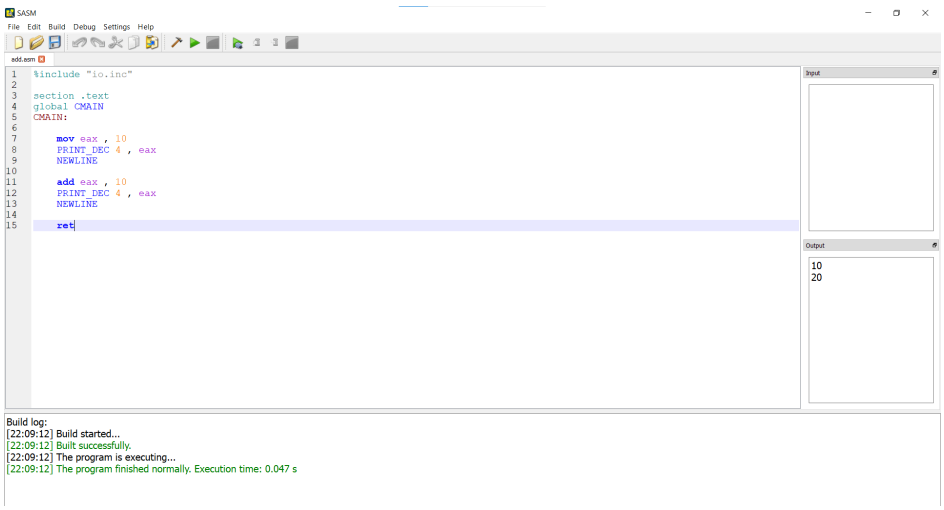
```
add destination , source
```

Here's the implementation:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    mov eax , 10
    PRINT_DEC 4 , eax
    NEWLINE

    add eax , 10
    PRINT_DEC 4 , eax
    NEWLINE
```

```
14
15      ret
```

Output



Here, we first copied the value 10 to the `eax` register and printed it. Then we issued the command `add eax , 10`. This command adds 10 to the value in `eax` register and stores the result in the `eax` register itself.
`10 + 10 = 20` , so the next string printed to screen is 20.

sub Command

As you have guessed, the `sub` command is used to perform the substraction operation.

The syntax is as follows:

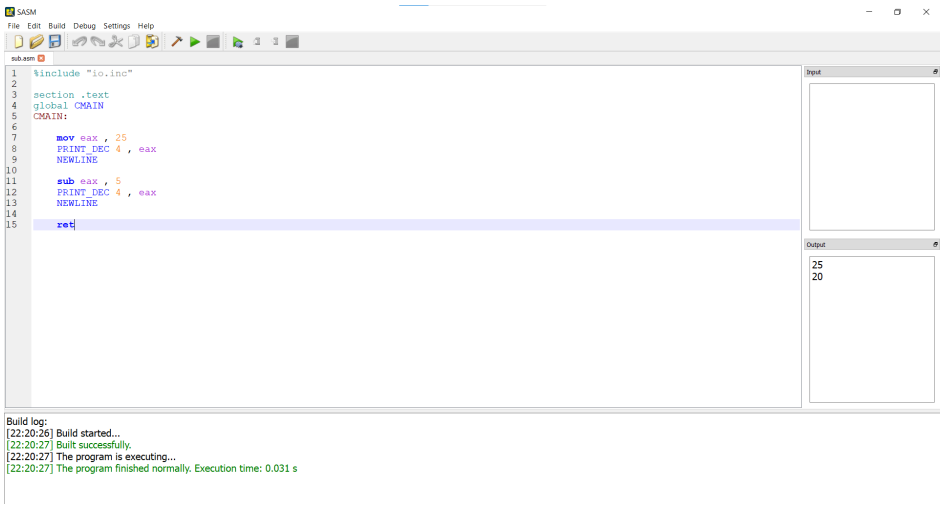
```
sub destination , source
```

Here's the implementation:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax , 25
```

```
 8      PRINT_DEC 4 , eax
 9      NEWLINE
10
11      sub eax , 5
12      PRINT_DEC 4 , eax
13      NEWLINE
14
15      ret
```

Output



The `sub` instruction here will substract 5 from 25 and the value 20 is printed.

push And pop Commands

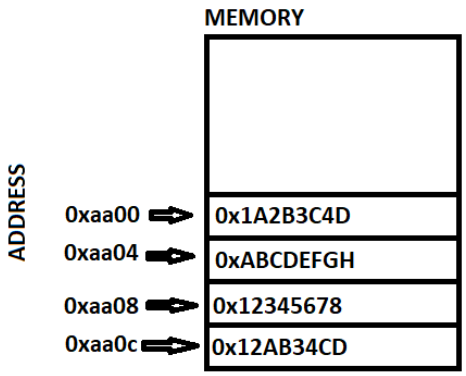
Working With Stack

Stack is an area in the main memory(RAM) where we could store variables and other data items. The two operations that we could do on stack memory is the `push` and `pop` operations. `Push` basically means to store a value in memory and `Pop` means to take a value from memory.

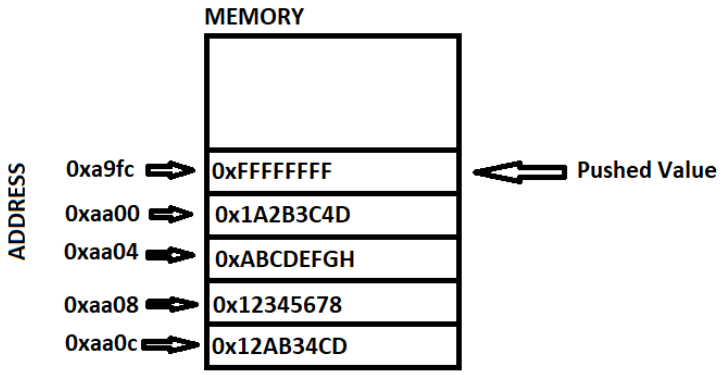
On the x86 architecture, pushing a value to stack means to put a value in lower memory address. When we attempt another push operation, The processor will put the value specified to it in memory just BEFORE the previous one. This means that the stack grows to lower memory addresses.

Popping values(Or taking value) from memory means taking value from the lowest address of the stack. Another pop attempt will take the value just AFTER the previous one. This means that pop instruction goes to higher memory addresses.

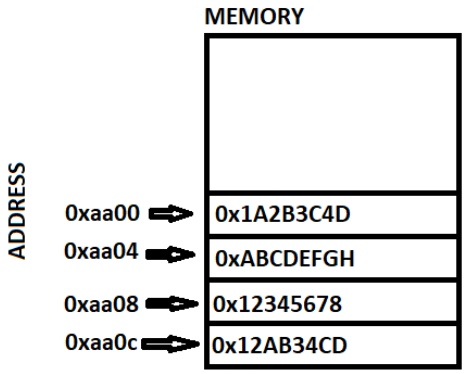
We will see an example image, Here's an image of the stack with some initial values in it:



Here is the image of the stack memory when a value `0xFFFFFFFF` is pushed onto it:



At this stage, If we issue a `pop` command, The stack memory will look like this:

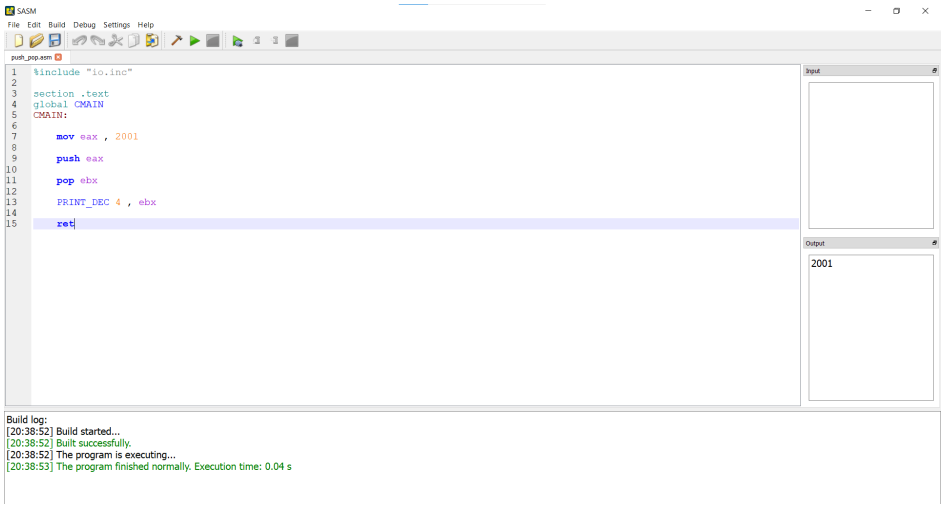


Practical Implementation Of Stack

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
```

```
 7      mov eax , 2001
 8
 9      push eax
10
11      pop ebx
12
13      PRINT_DEC 4 , ebx
14
15      ret
```

Output



Here , We first initialized `eax` with `2001`. Then we pushed the value in `eax` to the stack with the command `push eax`. This will place the value 2001 to the stack memory.

Then we issued the following command, `pop ebx`. This will take the value that we last pushed to the stack to the `ebx` register. So now, the `ebx` register have the value 2001.

Then we printed the value in `ebx`. This will print the value 2001 to the screen.

This is what the push and pop command basically does.

But how does the processor know what the last item pushed to the stack is?

This is where the `esp` register comes. The `esp` register holds address to the last item in stack memory.

For eg: If we push a value in `eax` register to the stack, What the processor does is first decrement the value in `esp` by four bytes(Which is the size of the `eax` register), Then it pushes the value in `eax` register to the address pointed by the `esp` register.

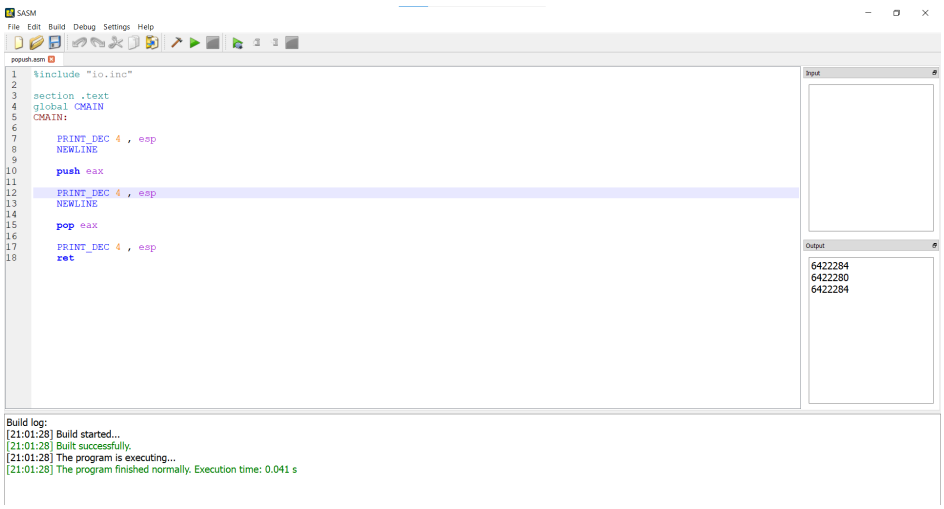
Likewise, when we pop a value to a register for eg: to the `ebx` register, The processor first takes the value in memory pointed by the `esp` register and copy it to register `ebx` and increases the value in `esp` by four bytes(Which is the size of `ebx` register).

So at conclusion, The `push` command decreases the value in `esp` register and the `pop` command increses the value in `esp` register.

Lets see it practically:

```
1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      PRINT_DEC 4 , esp
8      NEWLINE
9
10     push eax
11
12     PRINT_DEC 4 , esp
13     NEWLINE
14
15     pop eax
16
17     PRINT_DEC 4 , esp
18     ret
```

Output



Here, We First printed the initial value of the `esp` register and we got the value `6422284`. This is the address of the data which is at the top of stack.

Then we pushed the value in `eax` register to the stack with the `push eax` command.
This will practically decrease the value in esp register by four bytes. To confirm this, We printed the value in `esp` register after that `push` instruction.

And we got this value : `6422280`.

We can clearly see that the value decreased by four bytes. This value is the address of data at top of the stack.

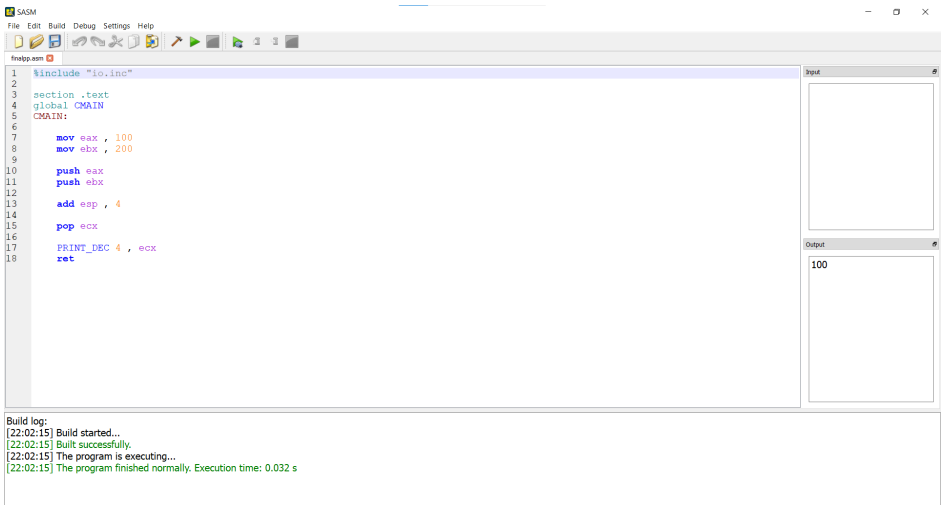
Finally we issued the `pop` command like this: `pop eax`. This will copy the value at top of the stack to the `eax` register and also increases the value in `esp` register by four bytes and we printed it. The value of `esp` register now is `6422284`. This is how the stack and commands to work with stack works.

Let's look at another example:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax , 100
```

```
 8      mov ebx , 200
 9
10      push eax
11      push ebx
12
13      add esp , 4
14
15      pop ecx
16
17      PRINT_DEC 4 , ecx
18      ret
```

Output



Here, We first moved the number `100` to the `eax` register and then moved the value `200` to the `ebx` register.

Then we pushed the value in `eax` and after that we pushed the value in `ebx` register to the stack.

At this stage, The value at the top of the stack is `200`(Value of `ebx` register). And if we issue a `pop` command and print the value, We will get `200` as output.

But before popping and printing the value, We Increased the value in `esp` register by four bytes with the command : `add esp , 4`. Now, As you have guessed, The esp register points to the value `100` which we first pushed.

And if we pop the value at top of stack with the `pop` command and print it, We will get `100` as output, not `200`.

You could try removing the `add esp , 4` line and differentiate the results.

I showed this example so that you could understand how all of the things incorporate.

pushAll And popAll Commands

What if you want to store values in all register to stack temporarily so that you could load new values to those register and later assign the previous values from stack back to registers?

We have some special commands to do that in x86 assembly.

To temporarily store the values in 32 bit general purpose registers to the stack, You could use the command `pushad` and to retrieve the value back to registers, You could use the `popad` command.

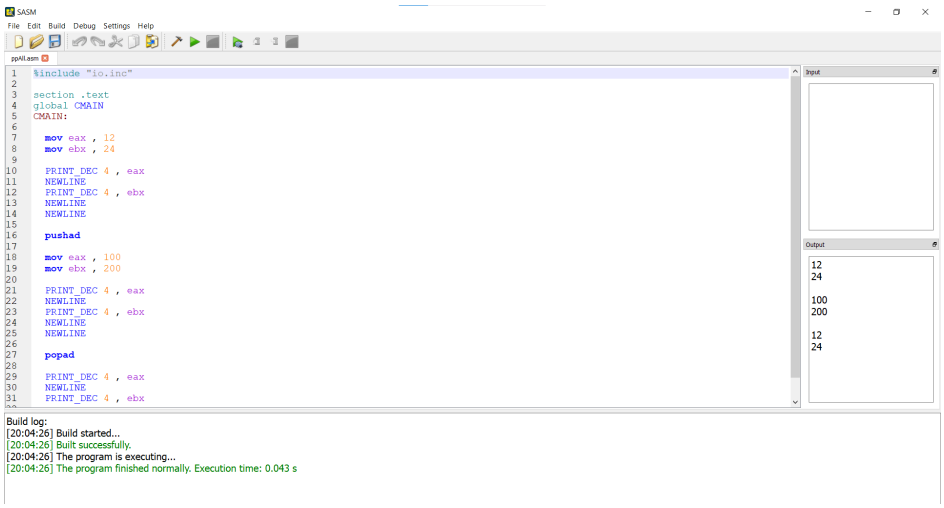
When working on 16 bit register, You could use `pusha` and `popa` commands for these operations.

Have a look at an example:

```
1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7    mov eax , 12
8    mov ebx , 24
9
10   PRINT_DEC 4 , eax
11   NEWLINE
12   PRINT_DEC 4 , ebx
13   NEWLINE
14   NEWLINE
15
16   pushad
17
18   mov eax , 100
19   mov ebx , 200
20
21   PRINT_DEC 4 , eax
```

```
22    NEWLINE
23    PRINT_DEC 4 , ebx
24    NEWLINE
25    NEWLINE
26
27    popad
28
29    PRINT_DEC 4 , eax
30    NEWLINE
31    PRINT_DEC 4 , ebx
32
33    ret
```

Output



I believe you could describe it yourself. But i will say how it works:

Here, We first moved some values to both the `eax` and `ebx` register and then we printed it. Now comes our command in focus: We put the `pushad` command there, This will push the values in all general purpose registers to the stack. Practically the values in `eax` and `ebx` registers also will be pushed.

Then we moved the value 100 to the `eax` register and 200 to `ebx` register and printed it. Here, the previous values in `eax` and `ebx` register got overwritten by new values.

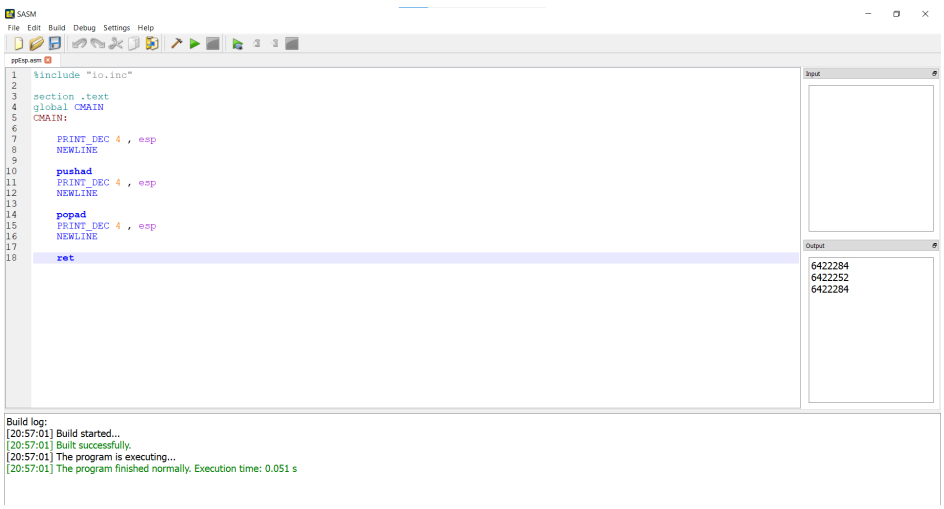
Then we issued the `popad` command. This command will copy the previous values we pushed to the stack with `pushad` command, Back to registers. So

the values we copied to `eax` and `ebx` registers before the `pushad` command also will be back to those registers.

See the following code:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    PRINT_DEC 4 , esp
    NEWLINE

    pushad
    PRINT_DEC 4 , esp
    NEWLINE

    popad
    PRINT_DEC 4 , esp
    NEWLINE

    ret
```

Output



Here, We first printed the initial value of `esp` register. Then we issued the `pushad` command. This will push whatever data is present in the general registers to the stack. We confirmed it by printing the value of `esp` after the

`pushad` operation, We saw that the value of `esp` decreased from that of the initial value.

Then, We issued the `popad` command, This will pop the values pushed to the stack back to registers, We confirmed it by printing the value in `esp` and we saw that the value came back to the initial value.

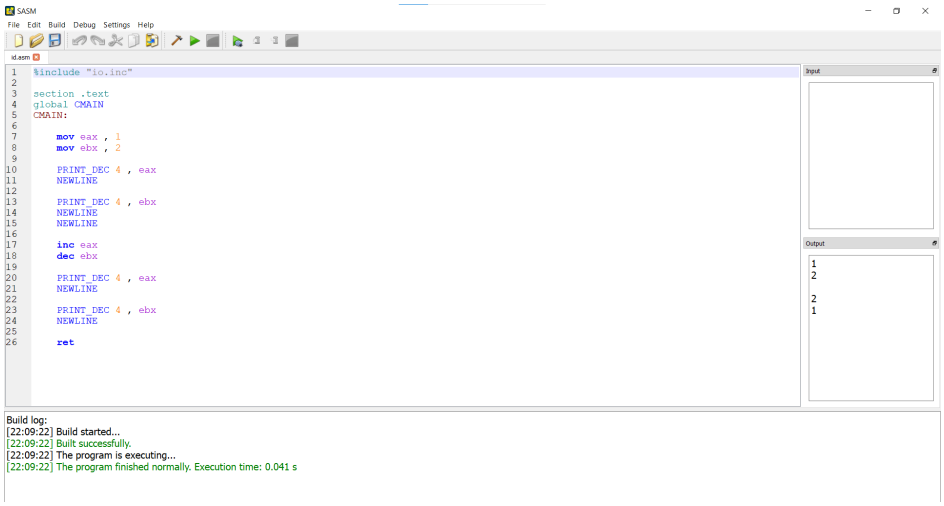
inc And dec Commands

Both `inc` and `dec` commands are simple. The `inc` command increments the value in a register by `1` and the `dec` command decrements the value in a register by `1`

Have a look at the following code:

```
1  %include "io.inc"
2
3  section .text
4  global CMAIN
5  CMAIN:
6
7      mov eax , 1
8      mov ebx , 2
9
10     PRINT_DEC 4 , eax
11     NEWLINE
12
13     PRINT_DEC 4 , ebx
14     NEWLINE
15     NEWLINE
16
17     inc eax
18     dec ebx
19
20     PRINT_DEC 4 , eax
21     NEWLINE
22
23     PRINT_DEC 4 , ebx
24     NEWLINE
25
26     ret
```

Output



The output itself explains all!!

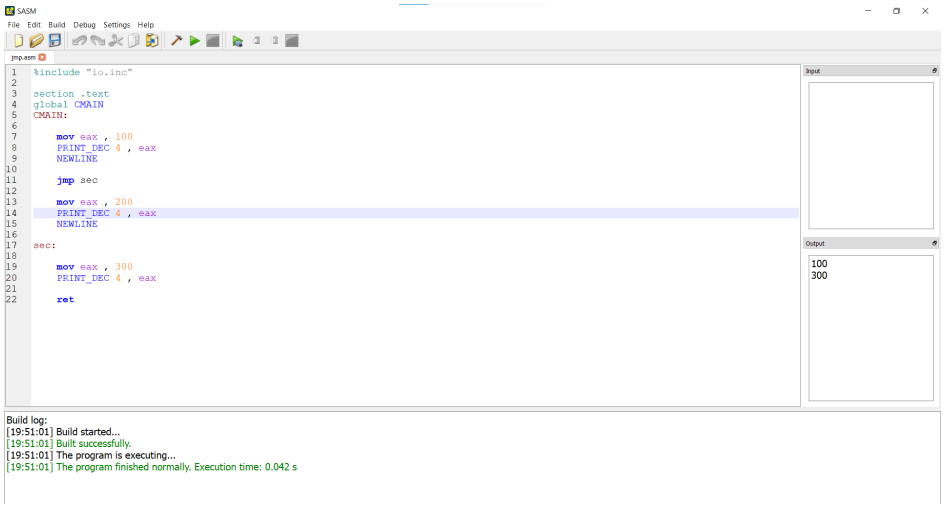
Extra Commands

jmp Command

`jmp` is just a command used to jump to a specified section of code, Let's see an example:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    mov eax , 100
    PRINT_DEC 4 , eax
    NEWLINE

    jmp sec

    mov eax , 200
    PRINT_DEC 4 , eax
    NEWLINE

sec:

    mov eax , 300
    PRINT_DEC 4 , eax

    ret
```

Output



Here, We first moved the value `100` to the `eax` register and printed it to the screen along with a newline. Then we issued the command `jmp sec` which transfers execution control to the `sec` section.

The `sec` section is defined with the line `sec:` , When cpu executes the command `jmp sec`, It jumps to the first code in the `sec` section. This results in the execution of code which moves the value `300` to the `eax` register and prints it.

You also saw a part of code which tries to moves the value `200` to the `eax` register and print it, But practically, That code won't be executed as we jumped to another section of code before reaching that line.

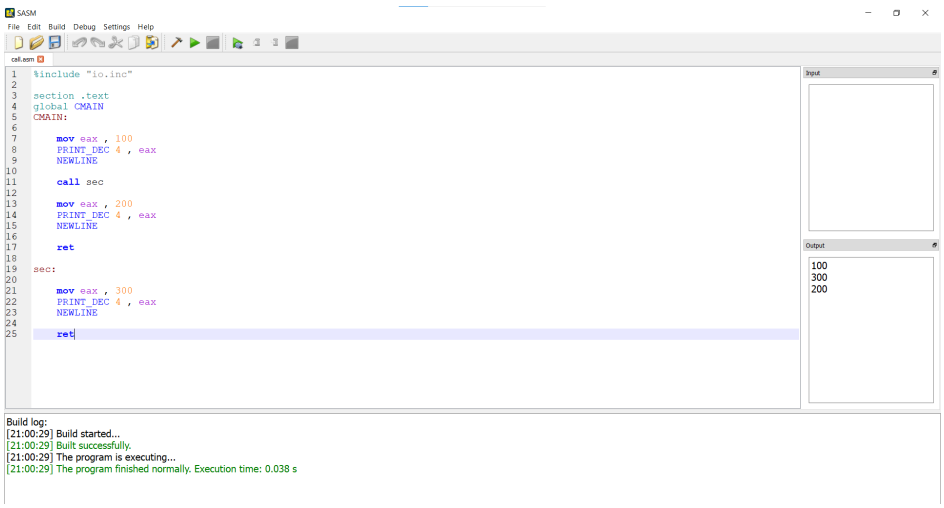
The thing to note here is that, if you want to create a new section, you need to suffix the section name with `:` , which means that if you want to create a section named `ADD` , You need to define it as `ADD:`. And to jump to that section, you need to call it like `jmp ADD`.

call Command

`call` is also a command which could be used to jump to other section but with an added capability. Let's make a small modification to the code we wrote for the `jmp` command :

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    mov eax , 100
    PRINT_DEC 4 , eax
    NEWLINE

    call sec

    mov eax , 200
    PRINT_DEC 4 , eax
    NEWLINE

    ret

sec:

    mov eax , 300
    PRINT_DEC 4 , eax
    NEWLINE

    ret
```

Output



In this code, We first moved the value `100` to `eax` register and printed it. Then, we took another approach to jump to the `sec` section using the `call` instruction.

Here, the difference from the `jmp` instruction is that, Before jumping to the section named `sec`, The cpu pushes the address of instruction after the `call` instruction to the stack before jumping to the `sec` section.

In our code, When the cpu executes the call instruction, It first pushes the address of the instruction `mov eax , 200` to the stack and jumps to the `sec` section.

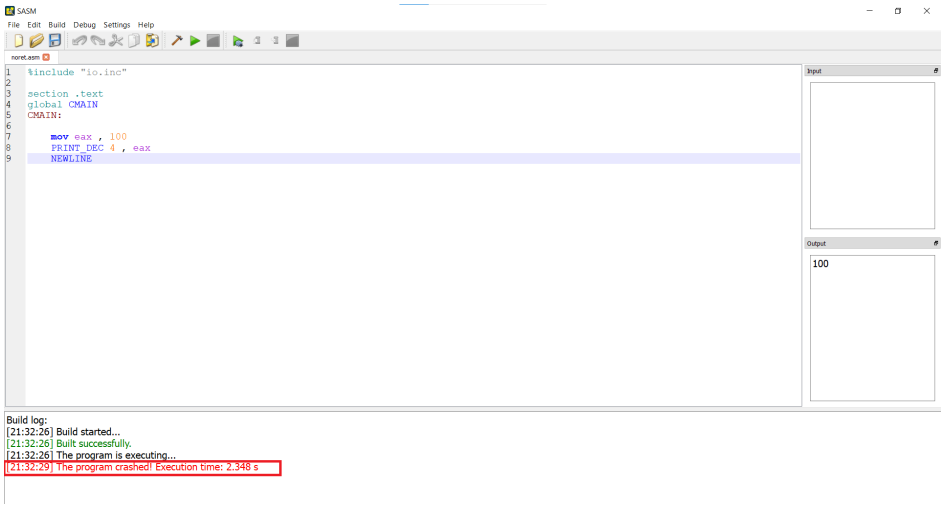
In that section, We printed the number `300` to the screen.
The last code in that section is `ret`, The `ret` instruction always pops the value at the top of the stack and stores it in the `eip` register(This is the register where the location of next instruction to be executed is present).

As the `call` instruction executed before, pushed an address to the stack, That pushed address itself will be popped to the `eip` register when the `ret` instruction is met. This results in the execution of the instruction `mov eax , 200` and then the printing of that number along with a newline. The next instruction to be executed is also `ret`. When this code is executed, The cpu will jump to code section which called our code which we defined as `CMAIN:`. This will result in execution of code to terminate the process.

Have a look at the following code with no `ret` instruction and you can see that the program crashed:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7     mov eax , 100
8     PRINT_DEC 4 , eax
9     NEWLINE
```

Output



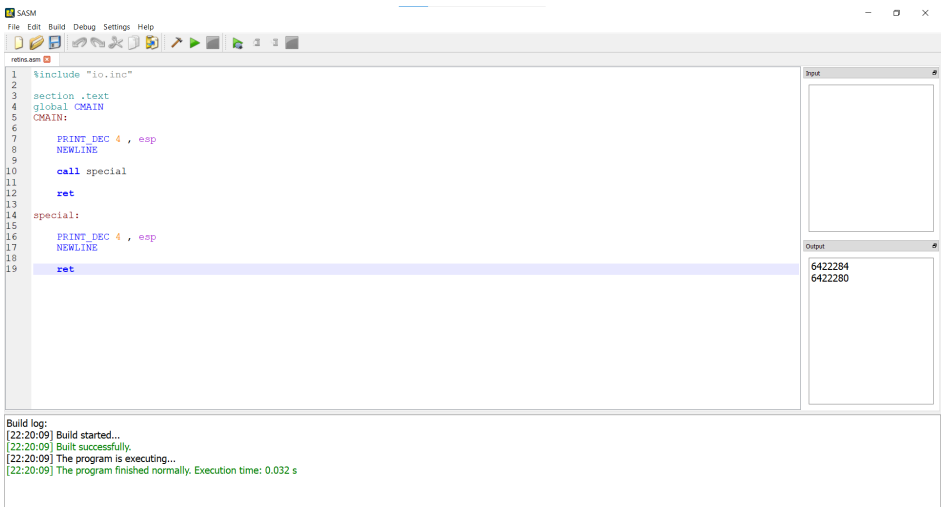
The actual reason for the crash is that as we haven't included `ret` instruction, The processor will start executing every data in memory after our code.

The processor will practically take everything in memory as executable code and the chance is that it might run code which it doesn't know or do unnecessary things.

Now, Look at the following code:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    PRINT_DEC 4 , esp
    NEWLINE

    call special

    ret

special:

    PRINT_DEC 4 , esp
    NEWLINE

    ret
```

Output



In this code, We first printed the value in `esp` register. This is the address of the data which is at the top of the stack.

Then, We called a section of code named `special` with the `call` instruction. This will push the address of the instruction after the `call` instruction to the stack and jumps to the code section named `special`.

In the `special` section, we printed the value in `esp` and you can see that the value changed. This is because the call instruction pushed an address to the stack. This changed value is the address of value at the top of the stack, which here is the address to instruction after the `call` instruction. After returning from the section `special`, The value in esp will be back to normal : You could try printing it.

I believe you have learned a lot till this. We will learn some more concepts before learning `OS-DEVELOPMENT`. Don't forget to have fun learning!!

cmp Command

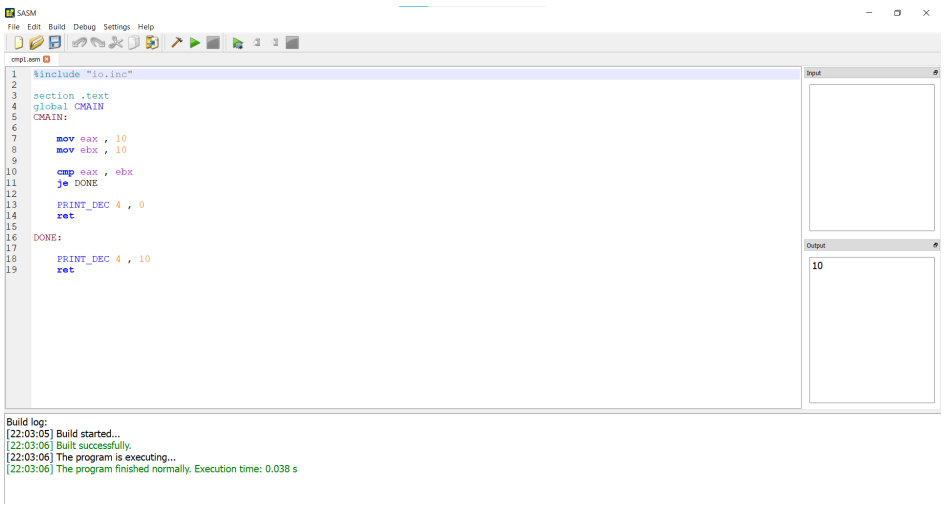
The `cmp` command is mostly used to compare two values to do a conditional execution.
The `if` command we used in `C` makes use of the `cmp` instruction.

The `cmp` instruction makes changes to some Flag registers so that a logical jump can be made with other instructions.

See the following code:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    mov eax , 10
    mov ebx , 10

    cmp eax , ebx
    je DONE

    PRINT_DEC 4 , 0
    ret

DONE:

    PRINT_DEC 4 , 10
    ret
```

Output



Here, We first copied the value `10` to both `eax` and `ebx` registers.
Then, With the command `cmp eax , ebx`, We compared the values in both the `eax` and `ebx` register.

The next command we issued is the `je` command. It stands for `Jump Equal` or `Jump If Equal`.

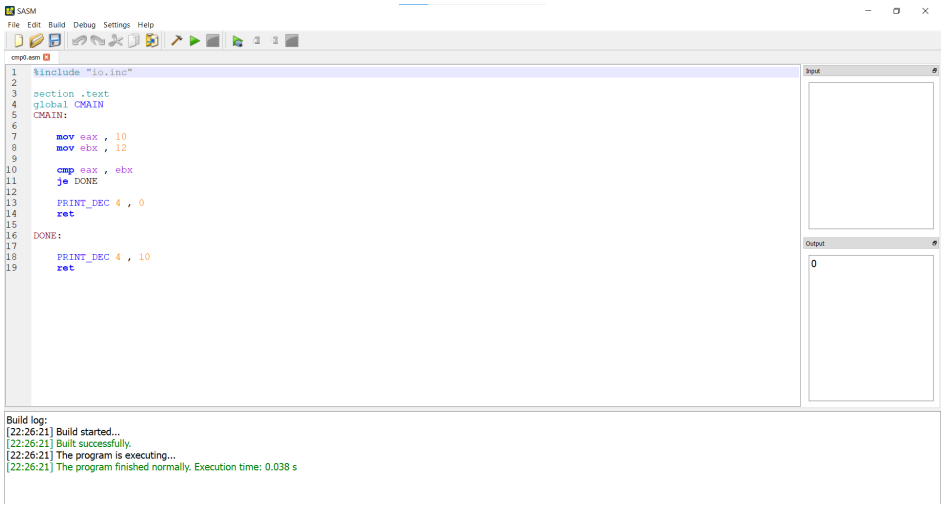
When executing the `je DONE` command, The processor checks the comparison we done using the `cmp` command and if the values in both `eax` and `ebx` registers are equal, Then it will jump to the code section named `DONE`.

If the values are not equal, It will continue execution of command from `PRINT_DEC 4 , 0`. This means that the cpu will not jump to the section `DONE` if the values are not equal.

Now look at the following code:

```
%include "io.inc"

section .text
global CMAIN
CMAIN:

    mov eax , 10
    mov ebx , 12

    cmp eax , ebx
    je DONE

    PRINT_DEC 4 , 0
    ret

DONE:

    PRINT_DEC 4 , 10
    ret
```

Output



Here, We applied only a small change, We moved the number `12` to the `ebx` register. So now, The values in `eax` and `ebx` registers are completely different.

So when executing the `je DONE` command, The cpu realises that values in those registers are completely different, So it will not execute the instructions in the `DONE` section and executes the instruction `PRINT_DEC 4 , 0` and finally terminates.

Here, We have used `je` as conditional operator, but we have more commands that can be used after the `cmp` command. They are:

1)**JE** : Jump Equal
2)**JNE** : Jump Not Equal
3)**JG** : Jump Greater
4)**JGE** : Jump Greater Than Or Equal
5)**JL** : Jump Less
6)**JLE** : Jump Less Than Or Equal

There are more branching commands to use with the `cmp` command, You could refer to other sources to learn more, But for this book, This much is enough.

I also recommend you to master x86 assembly. You could do it after completing this book or along with this book. It's all upto you.

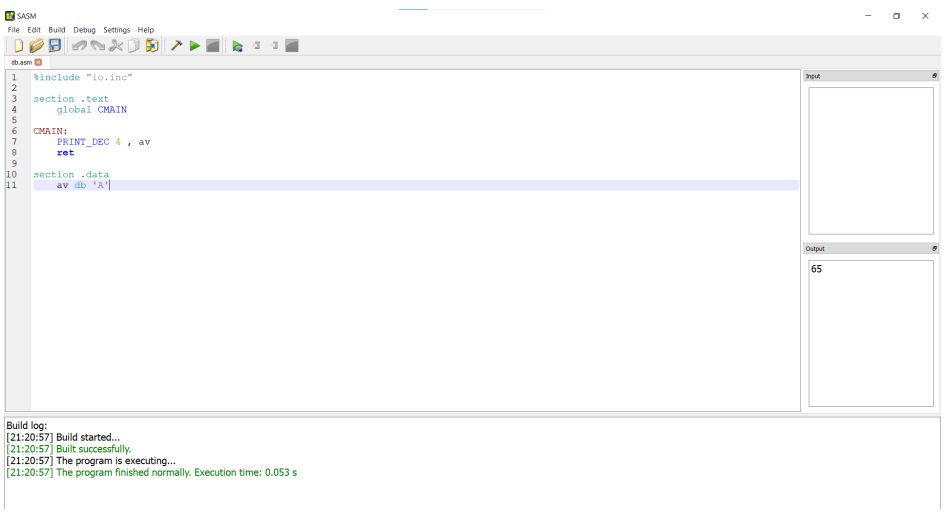
Variables

Nasm provides us with features to store variables in areas other that in the stack segment. **define directives** help us allocate these locations.

Have a look at the following:

```
1  %include "io.inc"
2
3  section .text
4      global CMAIN
5
6  CMAIN:
7      PRINT_DEC 4 , av
8      ret
9
10 section .data
11     av db 'A'
```

Output



Here you could see a section code starting with `section .data` and in that section, We declared a variable with `av db 'A'`.

Here `av` is the name of the variable, `db` says to allocate a byte and `'A'` will be assigned to the location reserved for `av`.

The output you saw on the screen is the ascii value of the character assigned to that variable.

You could also create an array of characters(String) with the `db` command itself.

Eg: `msg db 'Hello, world!', 0`

Here, some memory will be allocated to store the string `Hello, world!` ending with a null character(`0`). And it can be accessed by referencing with the variable name `msg`.

We have more keywords to allocate variables of different length. The full options are given below:

1)**db** : Allocates 1 byte
2)**dw** : Allocates 2 bytes(Word)
3)**dd** : Allocates 4 bytes(Doubleword)
4)**dq** : Allocates 8 bytes(Quadword)
5)**dt** : Allocates 10 bytes

Memory Addressing

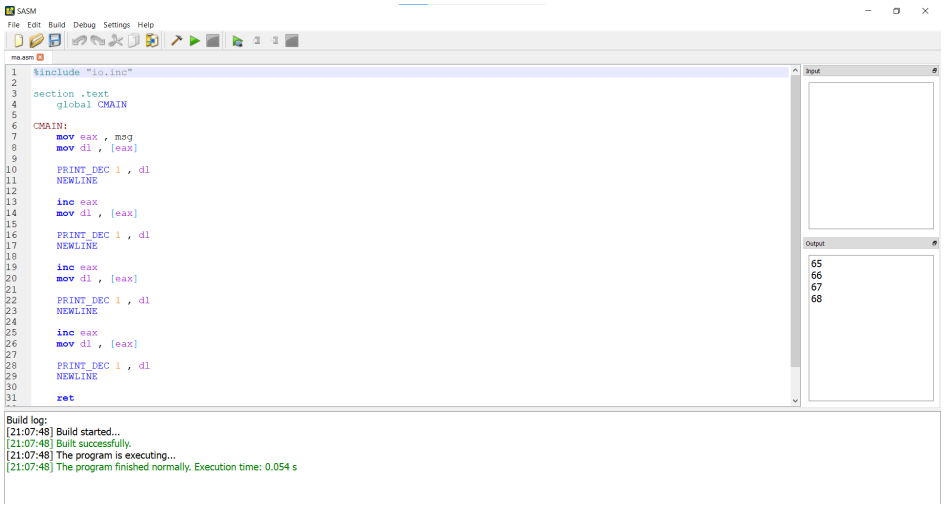
In the c programming language, We used pointers to access memory directly. Accessing memory directly is also possible in assembly and there need not be a doubt for that.

Lets see an example:

```
1  %include "io.inc"
2
3  section .text
4      global CMAIN
5
6  CMAIN:
7      mov eax , msg
8      mov dl , [eax]
9
10     PRINT_DEC 1 , dl
```

```
    NEWLINE

    inc eax
    mov dl , [eax]

    PRINT_DEC 1 , dl
    NEWLINE

    inc eax
    mov dl , [eax]

    PRINT_DEC 1 , dl
    NEWLINE

    inc eax
    mov dl , [eax]

    PRINT_DEC 1 , dl
    NEWLINE

    ret

section .data
    msg db 'ABCDEFGH'
```

Output



Here, At the very last of the program, We declared a variable named `msg` as:

```
msg db 'ABCDEFGH'
```

Whenever we reference something with the name `msg`, We will get the address of the string `ABCDEFGH`.

The very first line in the code is:

```
mov eax , msg
```

This line moves the address of the string `ABCDEFGH` to the `eax` register.

The next line is as follows:

```
mov dl , [eax]
```

Here, the Square Brackets in `[eax]` is used so that computer when at execution, will take the value in `eax` register as an address and take the data in that address to the `dl` register. So Here, Square Brackets are used as a sign to access memory.

The processor will only fetch one byte to the `dl` register as the size of `dl` register is only one byte.

Then we printed the value in `dl` register with the command `PRINT_DEC 1 , dl`.

This will print the ascii value of character `A`, as the first character in the variable `msg` is `A` itself.

Then we incremented address and printed each character's ascii values.

Here, We used the command `mov dl , [eax]` to copy the first character's ascii. We accessed the next value after incrementing the address with `inc` command. if we want to access the second character's ascii without using the `inc` command, You could use something like `mov dl , [eax + 1]`. This will give us the second character's ascii, likewise the command `mov dl , [eax + 2]` will give us third character's ascii. Further increment will give us further values in the `msg` variable.

Think of another case where we want to access the data in the address `61ff0c` specifically. Then, We could use the command `mov eax ,`

`[0x61ff0c]`. This will always move the value in address `61ff0c` to the `eax` register. But accessing a specific address like this wont work every time as when we are working on a pre made operating system(Windows here), the operating system may deny access to that memory which might crash the process. This is beacuse in windows and most other current operating systems, some memory parts are protected. But if the address we specify is not protected, We will get the data.

But when running the os **WE** made, We will get access to every locations as there is nothing preventing us from doing that. The kernel of the operating system have access to all memory locations.

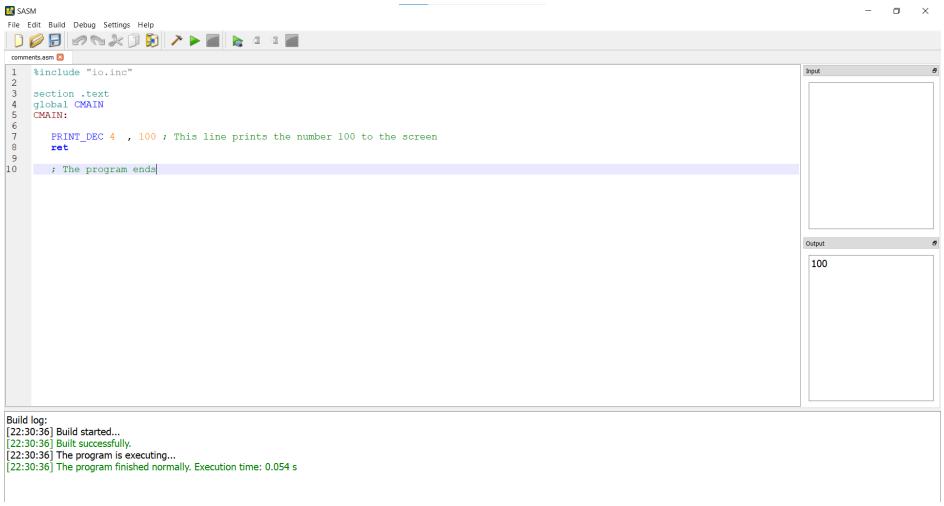
Comments

We could use the commenting feature in nasm to document our code, We could do it using `';'`

Example:

```
1 %include "io.inc"
2
3 section .text
4 global CMAIN
5 CMAIN:
6
7    PRINT_DEC 4  , 100 ; This line prints the number 100 to the screen
8    ret
9
10   ; The program ends
```

Output



Conclusion

Finally, You have completed everything to get started into developing your own os.
Now we will start learning operating system development, which you was eagerly waiting for. I think you have enjoyed this far, but the upcoming chapter are more interesting. Let's GOOOOOO!!!

Beginning Operating System Development

Introduction

We have travelled a long road till here. And we are now at the Main and Interesting topic. Before beginning, We have to know what an operating system is.

The early computers(At the very begining of computing), didn't have any operating system. In those days, computers are mainly used for mathematical calculation and for research purposes. As there is no operating system those days, The developers who want to solve a mathematical or other problems need to include extra code to handle memory, input/output devices etc along with the code to solve the problem they intend to solve.

As this is a repetitive job to include extra code along with the code to solve thier problem every time, They started the idea of `Operating Systems`. So, they made some code which manages memory , io devices etc and put it into the computers they ran. Later, If they need to solve a particular problem, They only need to focus on the code to solve the problem they want to solve as the `Operating System` they previously made will do all other tasks.

We all are lucky these days, We have plenty of operating systems to use and we have even writen programs for one. When writing and running your own operating system, It's you, who have the supreme power on your computer. You will manage all io devices and everything will be in your control. You could even write a GAME and run it on your computer with no operating system. You will get direct access to your video memory which lets you create your own graphics, Control it with keyboard and mouse and MAINLY with all of the processor power for your program only.

Anyway, We are going to start it, Lets Go!!

Writing Programs For Boot Sector

`Boot Sector` is the location in Hard Disk where we place code that we want to run. Most of the times, It contains code to load the operating system installed in that machine. But the code in boot sector is not the first code to get executed.

When we power on our computer, It starts copying some piece of code named `Bios` from `ROM`(Read Only memory) to main memory(`RAM`).

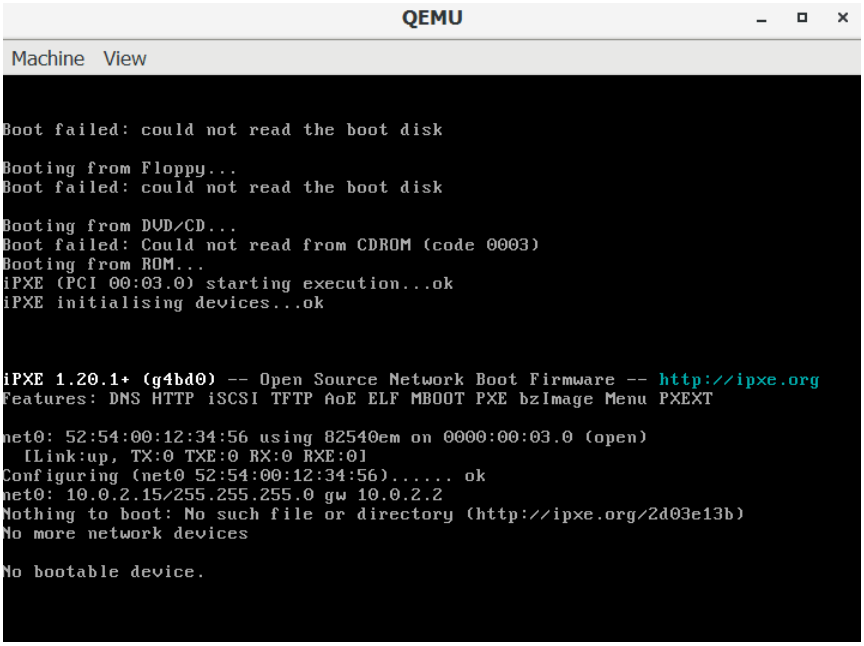
Then the cpu executes it. The main function of bios is to check if all connected devices are working properly and finally Load some code from the from first `512` bytes in the hard disk(Or #1 Bootable device) to Memory and execute it.

The location of this first `512` bytes is named as Boot Sector. We could write some programs and place it into the boot sector to execute it.

We are going to run our operating system in `QEMU` for now. Type the following command in cmd and hit enter:

```
qemu-system-i386
```

You will get a window like the following:



It says `no bootable device found`, But that's OK as we didn't give it with a file to boot, It will keep saying that.

Let's create a `512` byte file using `nasm` and give it to `QEMU` so that it could execute it.

First create a file named `boot-sect0.asm` and include the following code:

```
1 times 512 db 0
```

Here, what the code does is create a `512` byte long file and initialize all of the bytes with `0`. You already know this command: The `db` command(Or `define byte`).

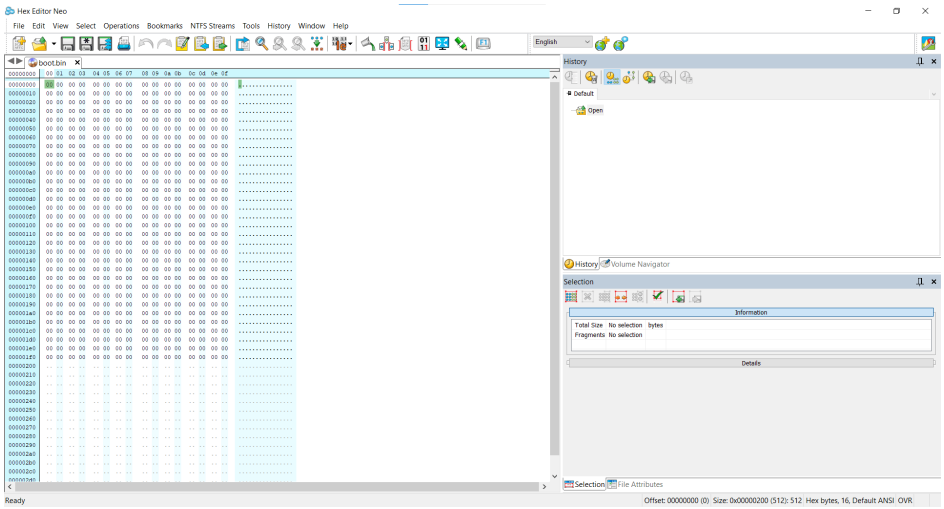
The `db` command defines a `byte`, But as we included the command `times 512` in `times 512 db 0`, It will define `512 bytes`, not `1 byte`. The `0` at the end is used to initializes all of the bytes with `0`.

assemble it using the following command:

```
nasm boot-sect0.asm -f bin -o boot.bin
```

In this command, `boot.bin` will be the name of the `512 byte` file which the assembler will create.

Open this file with the `Hex Editor` that we previously downloaded, You will see the following output:



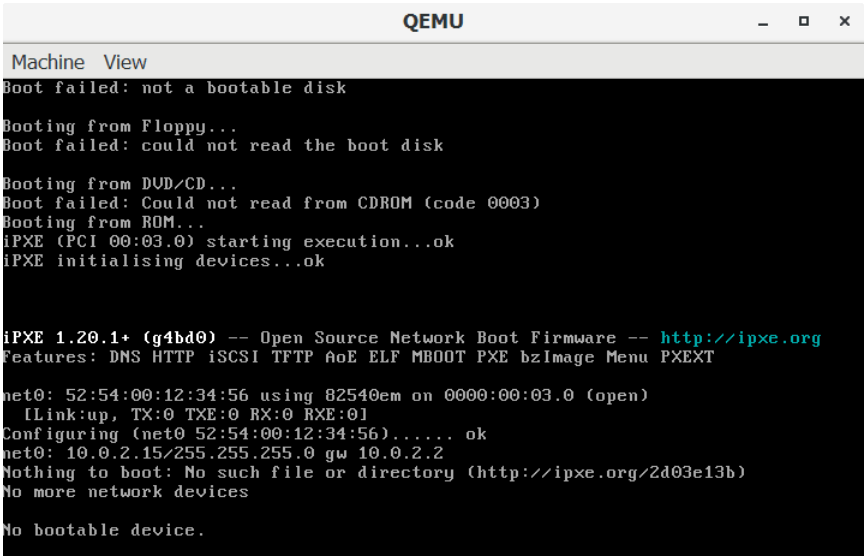
You could see that the file is `512` bytes long and all of the bytes are initialized with `0`.

Now, we could boot this file using `QEMU`. For that, Type the following command in cmd:

```
qemu-system-i386 -drive format=raw,file=boot.bin
```

Here, `boot.bin` is the name of file to be booted.

Output



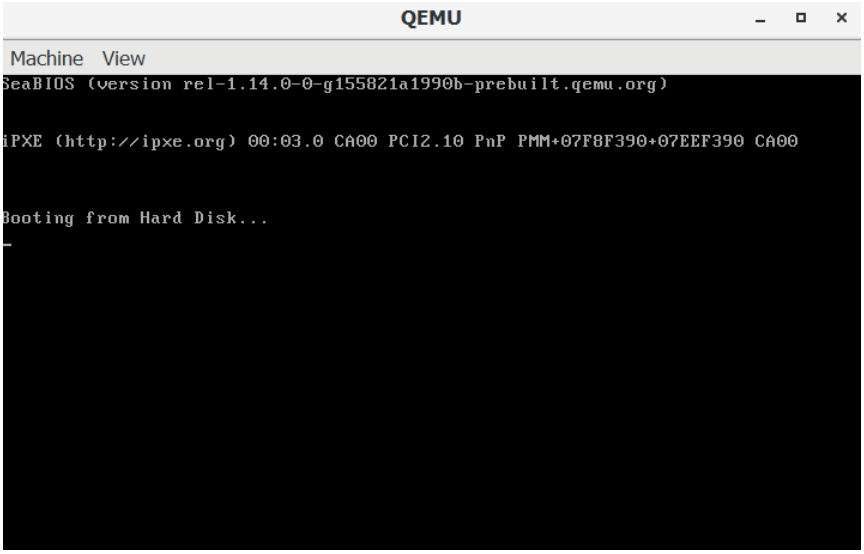
It is showing that no bootable device is found. There is nothing wrong here, It won't boot everything which is fed into it. For the computer to boot our operating system, It need to know whether the file is actually an executable. This is implemented so that, The computer will not try to take any piece of data as executable until we say that it is an executable.

We need to make small changes to the code so that the computer can confirm that it is an executable.

For that, put the following code into the source file and assemble it from cmd using nasm(Use the previous command which we used to assemble the source code using nasm):

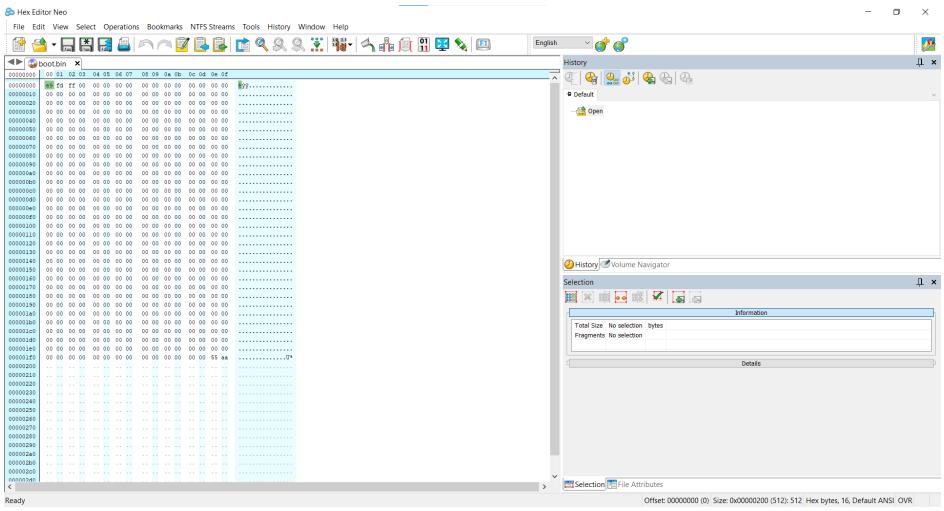
```
1 loop:
2       jmp  loop
3
4 times  510-($-$$) db 0
5 dw 0xaa55
```

Now, Boot the file with qemu, We will see the following:



Congratulations!!!!, You have successfully created something that the computer could direcly execute!!!

Lets learn how this works, Open the binary file we just booted, using the Hex Editor and look at its content.



Look at the very last two bytes here:

These bytes are called `Magic Number`, when our computer tries to boot from hard disk, It first checks if the last two bytes of the `512 byte` boot sector is

the this magic number or not, When it finds that the last two bytes are this magic number, It will start to execute the code from the 0'th location. In our case it executes the `jmp` instruction. This is an endless jump as it jumps every time to this jmp instruction itself so this program will not terminate.

We applied this `jmp` instruction here so that the computer will not execute the remaining bytes which we initialized it with `0`. There is a chance for the computer to crash as it executes the data that is not intended to be executed.

We applied the magic number with the command `dw 0xaa55`. This command will attach the bytes `aa` and `55` to the end of the file generated.

But when looking at the Hex Editor, we can see that the bytes are `55` and `aa`, not `aa` and `55`.

This is because of the `endianess` property we discussed in the assembly programming section. The x86 compilers and assemblers will put multi-byte data type in reverse order. This is what that happened here.

You can also see a section in our code : `times 510-($-$$) db 0`.

The job of this command is to fill the content between the bytes representing the `jmp` instruction and `dw` instruction with `0`.

It fills the content after the byte representation of the `jmp` instruction to the `510'th` byte with `0`.

The final `dw 0xaa55` command generates those bytes and places it to the very last of the file.

So finally, The size of the file becomes `512` bytes.

When creating further programs, We could replace the `jmp` instruction with whatever instruction we want, But we shouldn't take away the `dw` command which places the magic number at the last two byte locations. if we do that, The program won't be executed by the computer.

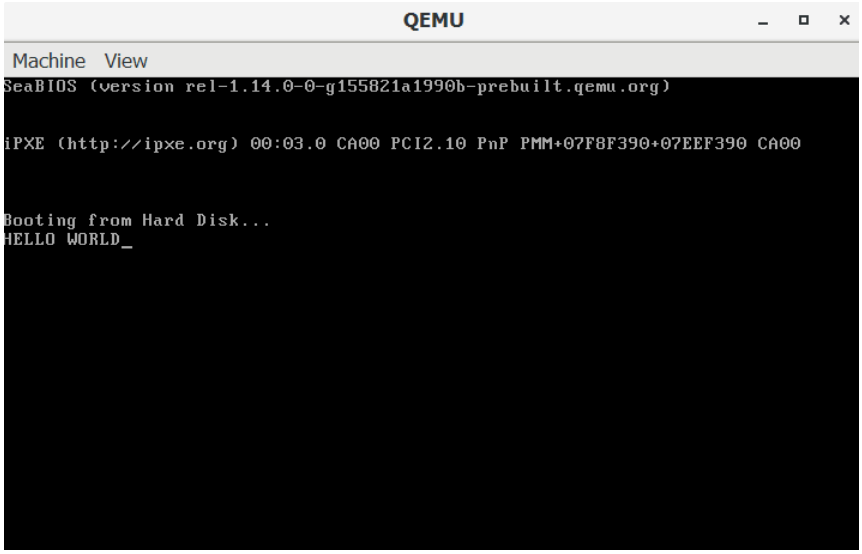
Printing To Screen (Hello , World OS)

So now, Lets try printing something to the screen.

Save and assemble the following code:

```
mov ah, 0x0e

mov al , 'H'
int 0x10

mov al , 'E'
int 0x10

mov al , 'L'
int 0x10

mov al , 'L'
int 0x10

mov al , 'O'
int 0x10

mov al , ' '
int 0x10

mov al , 'W'
int 0x10

mov al , 'O'
int 0x10

mov al , 'R'
int 0x10

mov al , 'L'
int 0x10

mov al , 'D'
int 0x10

jmp $

times  510-($-$$) db 0
dw 0xaa55
```

Boot the assembled file and we could see this:



We have successfully printed `HELLO WORLD` to the screen. Let's study the code:

`Please note that, the computer will be initially in 16 bit mode(16 Bit Real Mode) after boot-up. So, We could only use 16 bit registers such as ax , bx or lower parts of it such as ah , al , bh , dl etc..`

`We will later learn how to switch to 32 bit mode so that we could use 32 bit registers such as eax , ebx and also lower parts of it such as ax , bx etc..`

Here, we printed these characters by calling `BIOS` with `int 0x10`. The bios will do neccessary thing to print the characters to the screen.

We first copied the hex value `0e` to the ah register with the command : `mov ah, 0x0e`. This value is named as `The Scrolling Teletype`. Then we copied the character `H` to the `al` register using the command `mov al, 'H'`. Finally we called Bios using `int 0x10`

`int` is a command in assembly known as `Interrupt`. This command is used to pause the current execution and execute another code which is defined in a section named as `Interrupt Descriptor Table`.

`int 0x10` jumps to code section which have a list of display related routines. Bios selects which routine among them to be executed by looking at the values we passed to the registers.

When the Bios gets this interrupt, It first checks the value in `ah` register.

Here, as we copied the value `0e` to the `ah` register before the interrupt command, The bios realises that we want to implement a scrolling teletype. A scrolling teletype advances the cursor position to the next coloumn in the screen after printing a character.

The Bios knows which character to print to the screen by refering to the value in `al` register. As we previously passed the character `H` to `al` register, The bios prints the character `H` to the screen and advances the cursor to the next coloumn in screen.

After Bios prints that character, It passes the control of execution to the code we created. So practically, In our code, It will execute code which moves the character `E` to the al register and continue printing the character. With this method, we have printed the string `HELLO WORLD` fully to the screen.

The last command(`"jmp $"`) is and endless jump which jumps to that line itself.
This is so that the computer(As explained earlier), Will not try to execute data which is not intended to be executed.

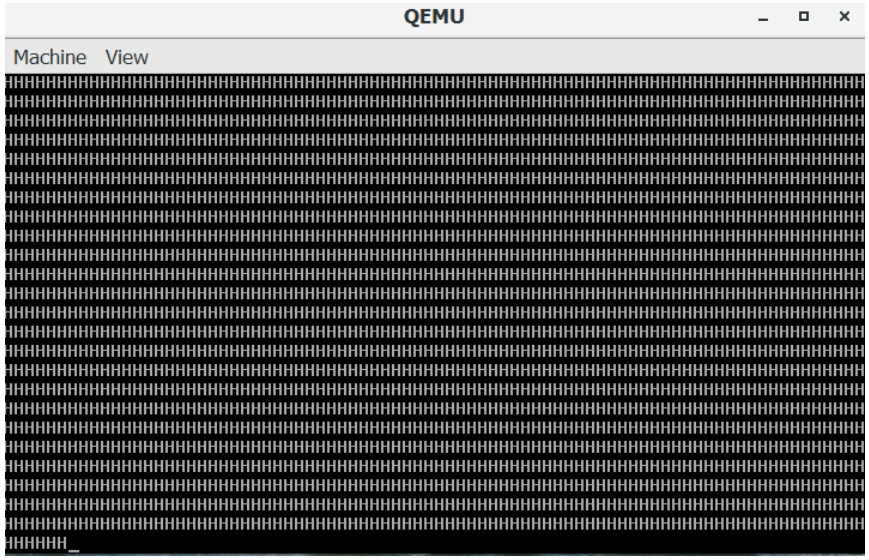
Filling The Screen With Characters(For Fun)!!

This section is just for FUN!!!, Lets fill the screen with some data.

I implemented it with the following code:

```
mov ah, 0x0e

Noend:

     mov al , 'H'
     int 0x10
     jmp Noend

times  510-($-$$) db 0
dw 0xaa55
```

Output



Filling The Screen With Colours

Bios also allows us to fill the screen with colours, Take a look at the following code:

```
mov ah , 0x0b
mov bh , 0x0
mov bl , 0xff
int 0x10

mov ah, 0x0e

mov al , 'H'
int 0x10

mov al , 'E'
int 0x10

mov al , 'L'
int 0x10

mov al , 'L'
int 0x10

mov al , 'O'
int 0x10

mov al , ' '
int 0x10

```

```
26 mov al , 'W'
27 int 0x10
28
29 mov al , 'O'
30 int 0x10
31
32 mov al , 'R'
33 int 0x10
34
35 mov al , 'L'
36 int 0x10
37
38 mov al , 'D'
39 int 0x10
40
41 jmp $
42
43 times  510-($-$$) db 0
44 dw 0xaa55
```

Output

Look at the first few lines of the above code:

```
mov ah , 0x0b
mov bh , 0x0
mov bl , 0xff
int 0x10
```

This few lines is the code which changed the color of screen. Try changing the value copied to `bl` register and you could see different colours.

You can goto the following link to get values for different colours.

```
https://en.wikipedia.org/wiki/BIOS_color_attributes
```

Other Bios Display Related Routines

This is an optional study. If you are interested, Please follow the following link to get information about other display related bios routines.

```
https://en.wikipedia.org/wiki/INT_10H
```

Running Programs Written In C

Switching To Protected Mode

Till here, We wrote programs in assembly language to interact with computer. Writing programs in assembly is both risky and time consuming. Writing the os in c is a good option(But some part of it always need assembly).

Before writing and running programs in C, We will switch our processor to `32 bit mode`(Or `32 bit protected mode`).

Switching to 32 bit mode allows us protect some memory locations from other user mode programs trying to access those location. If a user mode program could access these special locations where the Kernel of the os resides, Those programs could take control of the cpu so our OS would not have any control.

We can define which part of the memory area to be protected by defining it in a table named `GDT` or `Global Descriptor Table`. It is necessary to Set The GDT Before switching to 32 bit protected mode. After defining and loading the GDT, We could switch the cpu to 32 bit mode.

Defining The GDT

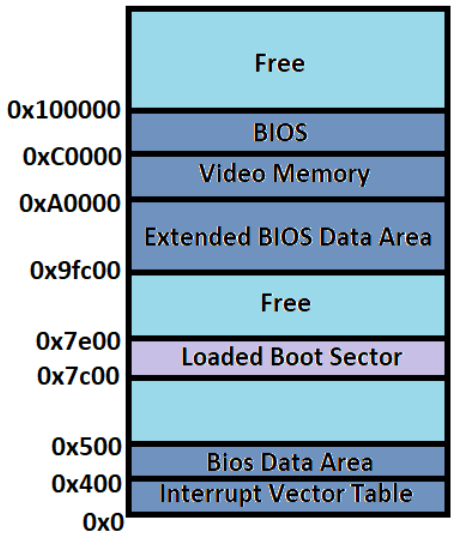
Life Without Bios

Before beginning this, Please know that we won't be able to use bios to do anything such as printing to screen after we define the GDT, Load it and finally Make the switch to Protected mode. We will make our own display related functions Later in this book.

Bios also gives other necessary functions such as for accessing i/o devices , Hard disks etc…. These too also will be unavailable after the switch.

Implementing The GDT

Lets see where our code and other resources will be present in memory:



From this image, You can see that the location where our Boot sector is present, is at `0x7c00`.

This means that Bios will load our Boot sector to `0x7c00` in memory.

Before switching the processor to 32 bit mode, We need to define our Global Descriptor Table(GDT).

GDT is a special data Structure which the processor directly validates.
It is little complex to understand for the first time. But this this can't be

avoided when trying to switch to protected mode. Let's First define GDT and switch to 32 bit mode:

Making The Switch

The Following code will define the gdt and Make the switch to 32 bit mode: (Alternatively, You can go to `https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/GDT.asm` to get a fine copy of this code.)

```
[org 0x7c00]

[bits 16]

;Switch To Protected Mode
cli ; Turns Interrupts off
lgdt [GDT_DESC] ; Loads Our GDT

mov eax , cr0
or  eax , 0x1
mov cr0 , eax ; Switch To Protected Mode

jmp  CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
;Forces the cpu to flush out contents in cache memory

[bits 32]

INIT_PM:
mov ax , DATA_SEG
mov ds , ax
mov ss , ax
mov es , ax
mov fs , ax
mov gs , ax

mov ebp , 0x90000
mov esp , ebp ; Updates Stack Segment




jmp $ ;Hang , Remove This To Start Doing Things In 32 bit Mode




GDT_BEGIN:

GDT_NULL_DESC:  ;The  Mandatory  Null  Descriptor
     dd 0x0
     dd 0x0
```

```
44
45 GDT_CODE_SEG:
46      dw 0xffff               ;Limit
47      dw 0x0                  ;Base
48      db 0x0                  ;Base
49      db 10011010b    ;Flags
50      db 11001111b    ;Flags
51      db 0x0                  ;Base
52
53 GDT_DATA_SEG:
54      dw 0xffff               ;Limit
55      dw 0x0                  ;Base
56      db 0x0                  ;Base
57      db 10010010b    ;Flags
58      db 11001111b    ;Flags
59      db 0x0                  ;Base
60
61 GDT_END:
62
63 GDT_DESC:
64      dw GDT_END - GDT_BEGIN - 1
65      dd GDT_BEGIN
66
67 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
68 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
69
70 times 510-($-$$) db 0
71 dw 0xaa55
```

You can try assembling and running this, But it won't show any special output as what we have done is only a switch to 32 bit mode.

Let's now understand what this code does:

The very first line `[org 0x7c00]` Says to the assembler to make adjustments to the program so that it could calculate `0x7c00` as the address which it will be loaded to.

The second line `[bits 16]` says to produce 16 bit opcodes for instructions given below it.

The line `cli` turns the interrupt services off. We previously made some interrupts to bios with the `int` command. from now onwards, we will not be able to use that feature. Please note, not to avoid this instruction when switching to 32 bit mode.

Next, We loaded the GDT we defined at the bottom of the code with: `lgdt [GDT_DESC]`

Now, focus on the following line in the code we made:

```
GDT_BEGIN:

GDT_NULL_DESC: ;The Mandatory Null Descriptor


    dd 0x0
    dd 0x0
```

Both `GDT_BEGIN` and `GDT_NULL_DESC` gives us the address to that location which we could later refer.

The next two lines starting with `dd` defines two null pointer(0x0). This is a mandatory null descriptor when defining the gdt. These locations will be used by the processor when working on things with the gdt.

THE GDT IS A COMPLEX CONCEPT TO UNDERSTAND FOR THE FIRST TIME. AS THE MOTIVE OF THIS BOOK IS ONLY TO INTRODUCE OS DEVELOPMENT, WE WONT DELVE MORE INTO THIS TOPIC. THIS TOPIC MAY POSSIBLY DEMOTIVATE YOU.

THIS TOPIC IS INCLUDED ONLY BECAUSE IT CAN'T BE AVOIDED WHEN SWITCHING TO PROTECTED MODE.

YOU DON'T NEED TO WORRY ABOUT THE GDT WHEN YOU ARE STARTING THE JOURNEY. YOU CAN JUST COPY PASTE THIS CODE, WHAT WE MAINLY FOCUS IS AT THE PRACTICAL SIDE OF OS DEVELOPMENT SUCH AS ACCESSING HARD DISK, KEYBOARD, MOUSE, CREATING GRAPHCAL USER INTERFACE ETC AND MORE. WE WILL GO INTO THAT IN LATER CHAPTERS.

WHEN YOU THINK YOU WANT TO MASTER GDT, THE FOLLOWING ARTICLES WILL HELP:

```
    https://wiki.osdev.org/Global_Descriptor_Table
    https://wiki.osdev.org/GDT_Tutorial
```

Look at the following lines:

```
GDT_CODE_SEG:

    dw 0xffff ;Limit
    dw 0x0 ;Base
    db 0x0 ;Base
    db 10011010b ;Flags
    db 11001111b ;Flags
    db 0x0 ;Base
```

These lines tells to the cpu about the code segment. We defined the `base` , `limit` and some flags. This values could be referenced using the name `GDT_CODE_SEG`.

Now lets see how we defined the data segment:

```
GDT_DATA_SEG:

    dw 0xffff ;Limit
    dw 0x0 ;Base
    db 0x0 ;Base
    db 10010010b ;Flags
    db 11001111b ;Flags
    db 0x0 ;Base
```

We could refer to this content in memory with the name `GDT_DATA_SEG`.

The line `GDT_END:` helps us refer to that name later to get the ending location of this GDT Entry.

The next line is the `"Informer"` of the GDT Entry we defined:

```
GDT_DESC:

    dw GDT_END - GDT_BEGIN - 1
    dd GDT_BEGIN
```

The line `dw GDT_END - GDT_BEGIN - 1` stores the size of our GDT Entry. We have reduced `1` from `GDT_END - GDT_BEGIN`, This is because the size of GDT(According to the x86 cpu), is always one less than the real size.

The next line `dd GDT_BEGIN` Stores the beginning address of our gdt entry.

We passed the address of `GDT_DESC` to the `lgdt` command at top so that it could access the `size` and `beginning location` of our GDT.

The next two lines:

```
    CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
    DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
```

defines the `CODE SEGMENT` and `DATA SEGMENT` locations so that we could later refer to them. In the first line, the `equ` command assigns the Reslut of `GDT_CODE_SEG - GDT_BEGIN`(Substraction operation) to `CODE_SEG`. The same applies to the second line also.

Now, go to the top of the code at:

```
    mov eax , cr0
    or eax , 0x1
    mov cr0 , eax
```

This code sets the `cr0` bit. This will switch the processor to `32 bit protected mode`.

We finally need to implement a `far jump` to fully utilize 32 bit mode. We implemented it with `jmp CODE_SEG:INIT_PM`. This will clear the contents in

cache memory and jump to The section named `INIT_PM`. Here is the `INIT_PM` section:

```
[bits 32]
INIT_PM:

    mov ax , DATA_SEG
    mov ds , ax
    mov ss , ax
    mov es , ax
    mov fs , ax
    mov gs , ax
    mov ebp , 0x90000
    mov esp , ebp
```

We started it with the line `[bits 32]`. This tells to the assembler to generate `32 bit opcodes` for the instructions following it. This is necessary as we previously switched to 32 bit mode by setting the `cr0` bit.

later we used the `mov` command to move the address of data segment to all of the `Segment Registers`.

Finally we set the `esp` and `ebp` registers. This also is very necessary as the C Programs we later make will surely push values to the stack.

The final line `jmp $` is used to stop processor from executing further code. This is technically a jump to that `jmp` instruction itself.

`IT'S FINE IF YOU ARE NOT VERY CLEAR ABOUT THIS TOPIC. YOU WILL BE ABLE TO UNDERSTAND THESE LATER.`

Making Way For Running C Code

Now, as we switched to 32 bit mode, We could start writing our code in C, But first we need a way to call it from assembly. initially, Only 512 bytes are loaded to memory by bios. When we make a C program and compile it with the boot sector, It's size will always be greater that 512 bytes.

So we need to add code to the boot sector program which loads the remaining part of the code to memory. A BOOT LOADER is a piece of code in Boot Sector which loads the remaining part of the os to memory.

Let's make a Boot loader:

Making A Boot Loader

You could download the full code for this section and the next section(Calling Our C Kernel) from here :

https://github.com/TINU-2000/OS-DEV/tree/main/Beginning-OS-DEV/Entering%20c

ALSO NOTE TO INCLUDE ALL OF THE SOURCE FILES IN SAME FOLDER

Boot Loader is only a small program. So, we will be able to include the boot loader program with the GDT defining program we just made before.

Lets add some code to it:

Alternatively , you could go to the following link to get the specific code we are going to discuss in this section :

https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/Boot.asm

```
[org 0x7c00]
[bits 16]

;Boot Loader
mov bx , 0x1000 ; Memory offset to which kernel will be loaded
mov ah , 0x02   ; Bios Read Sector Function
mov al , 30     ; No. of sectors to read(If your kernel won't fit into 30 sectors , \
you may need to provide the correct no. of sectors to read)
mov ch , 0x00   ; Select Cylinder 0 from harddisk
mov dh , 0x00   ; Select head 0 from hard disk
mov cl , 0x02   ; Start Reading from Second sector(Sector just after boot sector)

int 0x13        ; Bios Interrupt Relating to Disk functions


;Switch To Protected Mode
cli ; Turns Interrupts off
```

```
18 lgdt [GDT_DESC] ; Loads Our GDT
19
20 mov eax , cr0
21 or  eax , 0x1
22 mov cr0 , eax ; Switch To Protected Mode
23
24 jmp  CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
25 ;Forces the cpu to flush out contents in cache memory
26
27 [bits 32]
28
29 INIT_PM:
30 mov ax , DATA_SEG
31 mov ds , ax
32 mov ss , ax
33 mov es , ax
34 mov fs , ax
35 mov gs , ax
36
37 mov ebp , 0x90000
38 mov esp , ebp ; Updates Stack Segment
39
40
41 call 0x1000
42 jmp $
43
44
45
46
47
48 GDT_BEGIN:
49
50 GDT_NULL_DESC:  ;The  Mandatory  Null  Descriptor
51      dd 0x0
52      dd 0x0
53
54 GDT_CODE_SEG:
55      dw 0xffff               ;Limit
56      dw 0x0                  ;Base
57      db 0x0                  ;Base
58      db 10011010b    ;Flags
59      db 11001111b    ;Flags
60      db 0x0                  ;Base
61
62 GDT_DATA_SEG:
63      dw 0xffff               ;Limit
64      dw 0x0                  ;Base
65      db 0x0                  ;Base
66      db 10010010b    ;Flags
67      db 11001111b    ;Flags
68      db 0x0                  ;Base
69
70 GDT_END:
71
72 GDT_DESC:
73      dw GDT_END - GDT_BEGIN - 1
74      dd GDT_BEGIN
```

```
75
76 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
77 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
78
79
80 times 510-($-$$) db 0
81 dw 0xaa55
```

Here, We added the bootloader at the very begining of the code:

```
mov bx , 0x1000 ; Memory offset to which kernel will be loaded
mov ah , 0x02 ; Bios Read Sector Function
mov al , 30 ; No. of sectors to read(If your kernel won't fit into 30 sectors , you may need to provide the correct no. of sectors to read)
mov ch , 0x00 ; Select Cylinder 0 from harddisk
mov dh , 0x00 ; Select head 0 from hard disk
mov cl , 0x02 ; Start Reading from Second sector(Sector just after boot sector)

int 0x13 ; Bios Interrupt Relating to Disk functions
```

This code will load 30 sectors(512 * 30 bytes) to the memory location `0x1000` which we copied to the `bx` register. This 30 sectors are excluding the boot sector(which bios loaded).

We requested for this service to the bios with the `int 0x13` interrupt.

Then, after the code where we switched to 32 bit mode, We called the code at location `0x1000` with the command `call 0x1000`.

This will result in the execution of code we just loaded to memory using our bootloader. We will see how to compile a C program and place it into the location `0x1000` in order to execute it.

`YOU SHOULD SAVE THIS CODE TO A FILE, LETS NAME IT: Boot.asm`

Calling Our C Kernel

In order to execute the c program we make, We must know where our entry c function is in memory. This may be hard to find as in the future, if me add

more code to our c program, the offset where the entry function will be present might change.

We could solve this problem by using the `linker` to calculate automatically where the entry function will be.

`NOTE: WE WON'T DISCUSS ANYTHING ABOUT LINKER SCRIPTS, WE WILL WORK ONLY ON THE COMMAND LINE`

Open a new file and add the following code:

`Alternate link :`

`https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/Kernel_Entry.asm`

```
1 START:
2 [bits 32]
3 [extern _start]
4       call _start
5       jmp $
```

`Save this file and name it Kernel_Entry.asm`

Here, we declared an extern function named `_start` with the command `[extern _start]`
When assembling this code, the assembler will leave a hint to the linker that there will be a function named `start`, in an outside source file which we will call with the code `call _start`.

There is an interesting matter here. We declared the function as `_start` and expect the linker to link with `start`.

This is very true in this case. for eg: If you want to call a function named `enter`, you should declare it as `_enter`.

As the name of our entry c function is `start`, we should call it as `_start`.

Now, we will make our c program, Open a new file and add the following code:

Alternate link :

https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/main.c

```
int start(){
      char* video_memory = (char*) 0xb8000;
      *video_memory = 'K';
}
```

Name this file as main.c

Now, We have a lot of job to assemble , compile and link all of these files. Typing lot of commands for every build is boring and time consuming. So we will include all of the commands to assemble , compile and link together in a `.bat` file.

After including all of the commands and saving it, We could double click on it and the build process will be done automatically.

For this, please create a folder named `bin` in the directory where our codes exist.

Now, include the following code in a new file named `compile.bat`(Don't save it in the `bin` folder, save it where our other programs are present):

Alternate link :

https://github.com/TINU-2000/OS-DEV/blob/main/Beginning-OS-DEV/Entering%20c/compile.bat

```
nasm Boot.asm -f bin -o bin\bootsect.bin
nasm Kernel_Entry.asm -f elf -o bin\Entry.bin

gcc -m32 -ffreestanding -c main.c -o bin\kernel.o

ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o

objcopy -O binary -j .text  bin\kernel.img bin\kernel.bin
copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image

qemu-system-i386 -drive format=raw,file=bin\os-image
```

Executing this file will Do the Build process and launch the final executable.

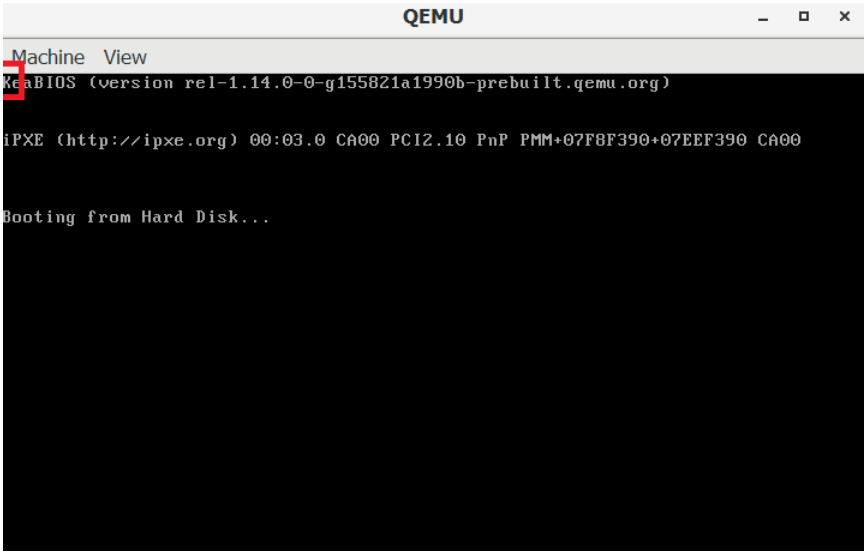
You could reuse this program every time. we will now be able to include more code to the c file, save it and finally click on the `.bat` file to automatically build and execute it.

All of the binary files will be generated to the `bin` folder

At the very last of our `.bat` file, we called qemu with: `os-image` as file to be booted.

Here, `os-image` is the final executable that we could distibute to others for use.

Run the `.bat` file and we will get the following output in `qemu`



The `K` at the top left of the screen ensures us that, the c program we made is working.

You can try changing the c program to print other characters.

The `Fun Begins` from the next chapter!!!!

NOTE: IT'S A GOOD IDEA TO CHECK IF THE .bat FILE WHEN LAUNCHED PRODUCES ANY ERROR MESSAGES TO THE TERMINAL. NO ERROR WILL BE PRESENT WHEN COMPILING THE ABOVE PROGRAM. BUT MAKE SURE TO CHECK WHEN SOME MODIFICATIONS ARE MADE.

Video Graphics

Introduction

Displaying something to screen is the crucial part when developing an operating system. As 90% of a computers output uses the Display system. Even during development, We may need to print the values in variables to debug any issue arising.

In this section, We are going to discuss about `Text Mode User Interface`. But this mode also allows printing colours to the screen.

Normally after the boot up, The computers video card will be in `Text Mode`. We could Print something to the screen by Poking memory location starting from `0xb8000`.

We could write directly to this address in memory to print characters and colours.

The Text Mode interface allows `80` characters wide and `25` characters lines per screen.
This means that the computer will be in `80 x 25` mode.

Let's learn how this works!!

```
NOTE: IT'S A GOOD IDEA TO USE A CROSS COMPILER WHEN
DEVELOPING YOUR OS, AS OUR C COMPILER TOOLSET MINGW LACKS
SUPPORT FOR RAW BINARY GENERATION HELPFUL IN WRITING OS.
WINDOWS LACKS A GOOD NATIVE CROSS COMPILER. SO IT WILL BE
PRACTICAL TO USE LINUX WHEN DEVELOPING THIS. LINUX GIVES
ACCESS TO GOOD CROSS COMPILERS. YOU COULD INSTALL ALL OF THE
MAIN TOOLS DESCRIBED IN THIS BOOK ON LINUX AND CONTINUE
DEVELOPMENT.
```

SOME PROBLEMS WILL OCCUR WHEN USING GOLOBAL VARIBLES ETC... BUT IN THIS BOOK, WE WILL ONLY USE METHODS THAT YOU COULD FOLLOW IN WINDOWS. BUT IN FUTURE, WHEN TRYING TO DO SERIOUS PROJECTS, CONSIDER USING CROSS COMPILERS ON LINUX SPECIFICALLY i386 cross compilers OR YOU COULD EVEN TRY "WINDOWS SUBSYSTEM FOR LINUX"

Poking Video Memory

Displaying Text and Colours To Screen

We saw that the computer will be initially in `80 x 25` mode. The video memory in text mode starts from `0xb8000`. Consider a situation where you want to print the character J to the Top Left of the screen, What you should do to implement this is to write the Ascii value of `J` To the location `0xb8000`.

In c, It will look like this:

```
char* aa = (char*) 0xb8000;
*aa = 'J';
```

Here we declared a character pointer named `aa` and assigned it with the address `0xb8000`. Then in the next line, We took the value in `aa` as an address and assigned it with the Ascii value of `J`. This will print the character J to the Top Left of screen. Now, if you want to add background and foreground colour to that character, You could poke the next memory location(`0xb8001`) to give that character a colour.

For representing the colour, The first four bits in the byte represents background colour and next four bits represent foreground colour.

To give the first character a colour, You can use the following c program:

```
char* aa = (char*) 0xb8001;
*aa = 0xb5;
```

Here, we copied the hex value `b5`. `b` will be the background colour and `5` will be the foreground colour.

This means that the video card uses two bytes for representing a single character.

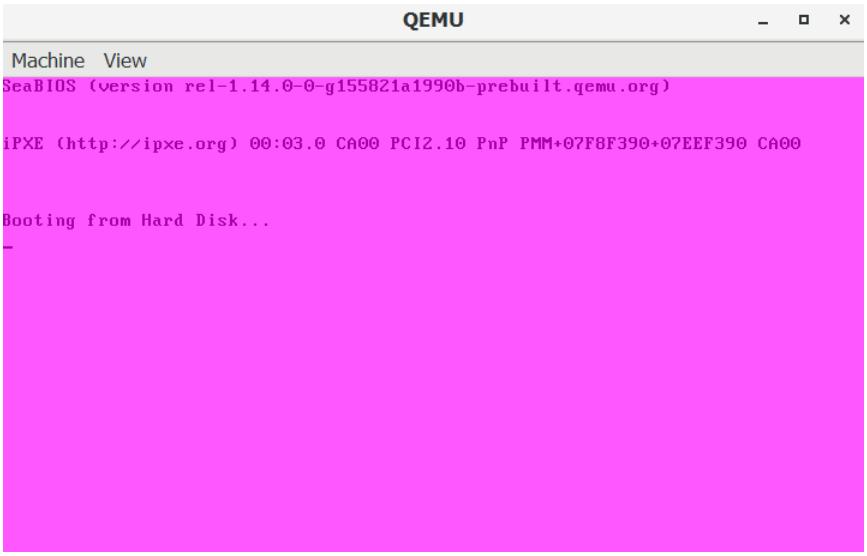
Suppose If you only want to Print the string `AS` to the screen with no colour. You could copy the character `A` to `0xb8000` and `S` to `0xb8002`.

So the total size of Text Mode Video Memory is `2 * 80 * 25 bytes`.

Lets see how to fill the screen with a special colour:

```
1 int start(){
2       char* cell = (char*) 0xb8000;
3       int i = 1;
4       while(i < (2 * 80 * 25)){
5               *(cell + i) = 0xd5;
6               i += 2;
7       }
8 }
```

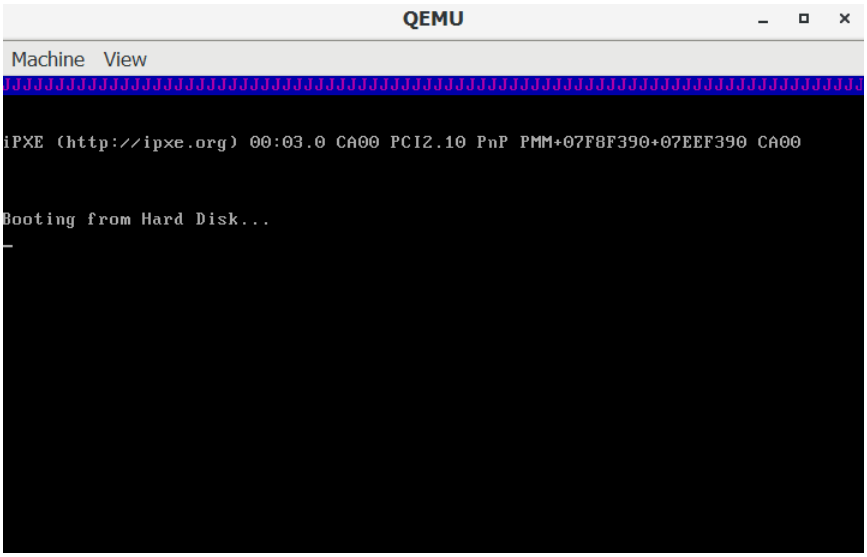
Save this program in the C Source file we created in the previous chapter. It will give the following output when executed:



Let’s now see an example where the first row of the screen fills with a character:

```
int start(){
      char* cell = (char*) 0xb8000;
      int i = 0;
      while(i < (2 * 80)){
              *(cell + i) = 'J';
              *(cell + i + 1) = 0x15;
              i += 2;
      }
}
```

Output



You could even make a `GAME` or `SMALL SCALE GUI` with this mode. In later chapters, we will learns how to switch to Graphics mode to display content to screen pixel by pixel.

Examples

You could skip this section if you want, This is just for fun!!!!

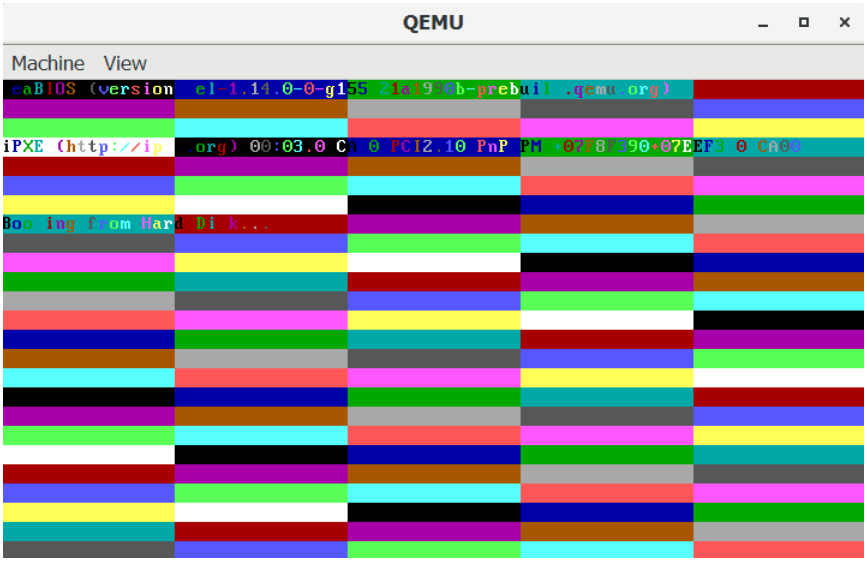
Alpha

Program:

```
int start(){
      char* TM_START = (char*) 0xb8000;
```

```
3       int i = 1;
4       char co = 0;
5       while(i < 2 * 80 * 25){
6               *(TM_START + i) = co;
7               i += 2;
8               co++;
9       }
10 }
```

Output:

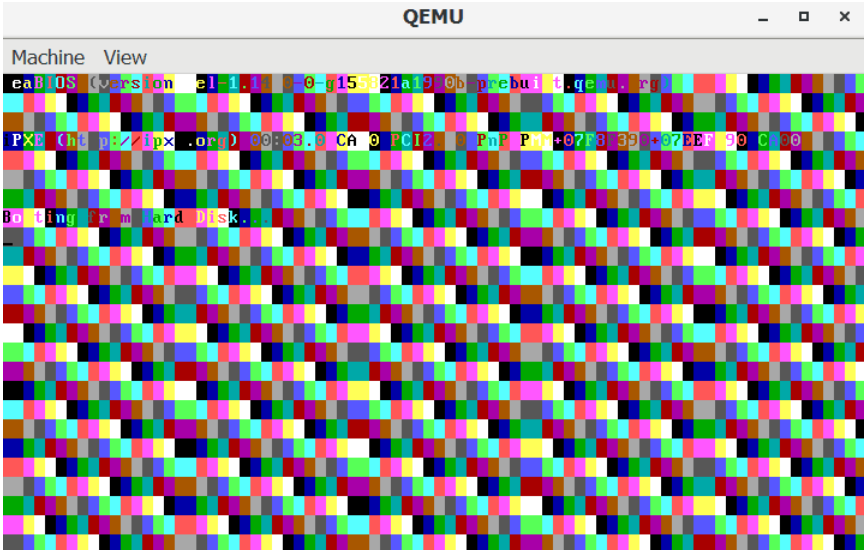


Beta

Program:

```
1 int start(){
2       char* TM_START = (char*) 0xb8000;
3       int i = 1;
4       char obj = 0;
5       while(i < 2 * 80 * 25){
6               *(TM_START + i) = obj;
7               i += 2;
8               obj += 15;
9       }
10 }
```

Output:



Gamma

Program:

```
1 int start(){
2       char* TM_START = (char*) 0xb8000;
3       int i = 1;
4       char obj = 0;
5       while(i < 2 * 80 * 25){
6               *(TM_START + i) = obj;
7               i += 2;
8               obj += 100;
9       }
10 }
```

Output:

Delta

Program:

```
1  int start(){
2        char* TM_START = (char*) 0xb8000;
3        int i = 0;
4        char obj = 0;
5        while(i < 2 * 80 * 25){
6                *(TM_START + i) = obj;
7                i++;
8                obj++;
9        }
10 }
```

Output:

Implementing Graphics Driver

In future, it's hard for us to print strings to the screen by looking at every details of Text Mode Video Graphics. So, we are going to implement our own "printf" function that we could later call to do the print operation.

Link to project files for this section :

https://github.com/TINU-2000/OS-DEV/tree/main/Video%20Graphics0

```
1  void cls();
2  void setMonitorColor(char);
3
4  void printString(char*);
5  void printChar(char);
6
7  void scroll();
8
9  void printColorString(char* , char);
10 void printColorChar(char , char);
11
12 void getDecAscii(int);
13
14 char* TM_START;
15 char NumberAscii[10];
16 int CELL;
17
18 int start(){
19     TM_START = (char*) 0xb8000;
20     CELL = 0;
```

```
21
22      cls();
23      setMonitorColor(0xa5);
24
25      char Welcome[] = "Welcome To OS0 : Copyright 2021\n";
26      char Welcome2[] = "Command Line Version 1.0.0.0\n\n";
27      char OSM[] = "OS0 > ";
28
29      printString(Welcome);
30      printString(Welcome2);
31      printColorString(OSM , 0xa8);
32  }
33
34
35  void cls(){
36      int i = 0;
37      CELL = 0;
38      while(i < (2 * 80 * 25)){
39              *(TM_START + i) = ' '; // Clear screen
40              i += 2;
41      }
42  }
43
44  void setMonitorColor(char Color){
45      int i = 1;
46      while(i < (2 * 80 * 25)){
47              *(TM_START + i) = Color;
48              i += 2;
49      }
50  }
51
52  void printString(char* cA){
53      int i = 0;
54      while(*(cA + i) != '\0'){
55              printChar(*(cA + i));
56              i++;
57      }
58  }
59
60  void printChar(char c){
61      if(CELL == 2 * 80 * 25)
62              scroll();
63      if(c == '\n'){
64              CELL = ((CELL + 160) - (CELL % 160));
65              return;
66      }
67      *(TM_START + CELL) = c;
68      CELL += 2;
69  }
70
71  void scroll(){
72      int i = 160 , y = 0;
73      while(i < 2 * 80 * 25){
74              *(TM_START + y) = *(TM_START + i);
75              i += 2;
76              y += 2;
77      }
```

```
78      CELL = 2 * 80 * 24;
79      i = 0;
80      while(i < 160){
81              *(TM_START + CELL + i) = ' ';
82              i += 2;
83      }
84  }
85
86  void printColorString(char* c , char co){
87      int i = 0;
88      while(*(c + i) != '\0'){
89              printColorChar(*(c + i) , co);
90              i++;
91      }
92  }
93
94  void printColorChar(char c , char co){
95      if(CELL == 2 * 80 * 25)
96              scroll();
97      if(c == '\n'){
98              CELL = ((CELL + 160) - (CELL % 160));
99              return;
100     }
101     *(TM_START + CELL) = c;
102     *(TM_START + CELL + 1) = co;
103     CELL += 2;
104 }
105
106 void getDecAscii(int num){
107     if(num == 0){
108             NumberAscii[0] = '0';
109             return;
110     }
111     char NUM[10];
112     int i = 0 , j = 0;
113     while(num > 0){
114             NUM[i] = num % 10;
115             num /= 10;
116             i++;
117     }
118     i--;
119     while(i >= 0){
120             NumberAscii[j] = NUM[i];
121             i--;
122             j++;
123     }
124     NumberAscii[j] = 'J';
125     j = 0;
126     while(NumberAscii[j] != 'J'){
127             NumberAscii[j] = '0' + NumberAscii[j];
128             j++;
129     }
130     NumberAscii[j] = 0;
131 }
```

Here, you could call the `printString` function by passing it with an address of null terminated character array. This will print the string to the screen.

`printColorString` can be called with the address of null terminated character array as first argument and a byte representing background and foreground colour as second argument.

`printChar` can be called with the character to be printed as the first argument.

`printColorChar` can be called with the first argument as the character to be printed and a byte representing background and foreground color as second argument.

These functions will also take care of the scrolling activity by calling the `scroll()` function.

We do not need to call the `scroll()` function from our side. These things will be handled by the printing functions.

if you need to turn an integer variable to a string which could later be used to print using the `printString` function, You could use the `getDecAscii` function. Pass it with the integer to be turned to string and later refer to the array `NumberAscii` to print it. Call the `printString` function with `NumberAscii` as argument to print the number generated by `getDecAscii` function.

Try printing some coloured and non coloured string!!

Output of the Program we developed for this section:

Developing a Simple Video Player

In this section, We are not going to discuss or implement modern type video players, But we will cover the basic concept of video playing.

Theory

`Videos` are actually moving pictures. When we see multiple pictures during a small amount of time, Our brain takes it as moving or gives us a sense of The so called `"Video"`.

What the computer do to present us with a video is by poking the video memory.
Changing the data contained in the video memory continuously gives us this feel.

But the processor is very faster than our video card and the monitor. So it won't be able to display every frame the processor generate very fastly. So the solution is to minimize the frame updating procedure depending on the refresh rate of the monitor.

But the computer must also be fast enough to play about `60` frames during a second.

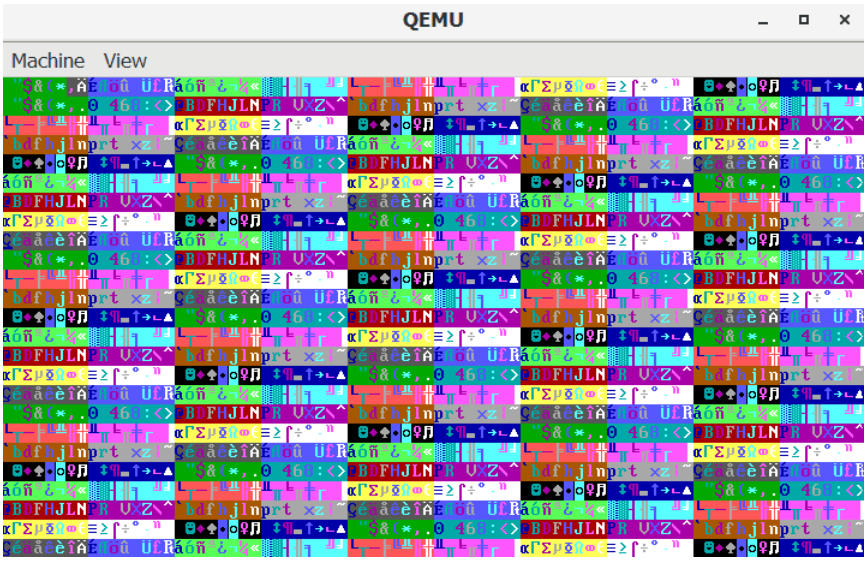
Human eyes see upto `60` frames per second. So theres no point in generating video with frames more that `60` per second.

Let's make a program which gives us with a `VIDEO` feel.

Practical Implementation

```
int start(){
      char* TM_START = (char*) 0xb8000;
      int i;
      char obj = 0;
      while(1){
              i = 0;
              while(i < (2 * 80 * 25)){
                      *(TM_START + i) = obj;
                      i++;
                      obj++;
              }
      }
}
```

One Frame Of Output During Video Playing:



The program is in an infinite loop with `while(1)`.

Implementing Keyboard Driver

Introduction

Keyboard is the primary input device for a computer. Working with keyboard is a non avoidable factor in os development.

How does the computer know when a key is being pressed?

This is an interesting question. We could try to implement it by asking keyboard everytime if there is any input or not. But this is not a good idea.

Keyboard is very slower than the processor. Asking it everytime will affect the processing speed badly. And it also will put lot of traffic in the system bus.

We need to avoid that. So the solution implemented by the x86 chip manufacturer uses interrupts.

When ever a key is being pressed, The cpu will call a fuction that we pre define to it. The processor won't do the keyboard logic, but it will let us execute some code whenever a key is being pressed.

We need to make some code to handle the keyboard input, and pass its address to the interrupt descriptor table and say to the processor to load it.

After loading all the parameters and the location of our keyboard handling code, It will call that code whenever a key is being pressed.

When we get this interrupt, We could try reading from the keyboard which gives us the key being pressed.

What keyboard gives as the value for pressed key is not ascii. It is named as `scan codes`. We need to write our own program to convert it to ascii or other text encoding schemes.

Scan Codes

As said earlier, we need to know the values given by keyboard(scan codes) in order to convert it to ascii.

The following link lists the scan codes:

`https://wiki.osdev.org/PS/2_Keyboard`

You can also take this link as an alternate way to know more about keyboard working

Implementing Keyboard Driver

Modern keyboards use `usb` interface to communicate to the computer. This section covers working of `PS/2` keyboards. But usb keyboards will also work here as usb keyboard emulates the older `PS/2` keyboards.

The PIC Chip

`PIC` or `Programmable Interrupt Controller` is a chip in the computer whose main job is to generate interrupts. When a key in keyboard is pressed, The Chip inside keyboard tells to the pic chip inside our computer to generate a #1 Interrupt. The pic chip will then decide the time to notify the cpu about the interrupt. When the cpu gets the message which says a key is being pressed, It executes a set of code which we told earlier to the cpu to execute when a key is pressed.

We can make use of the `in` assembly instruction to read the key being pressed.
It is very necessary to do this read as a failure in reading from keyboard will prevent the keyboard from giving further keyboard press events.

Practical Implementation

The full source code for this section could be downloaded from:

`https://github.com/TINU-2000/OS-DEV/tree/main/Keyboard`

Have a quick look at the code here:

`main.c`

```
#define PIC1_C 0x20
#define PIC1_D 0x21
#define PIC2_C 0xa0
#define PIC2_D 0xa1

#define ICW1_DEF 0x10
#define ICW1_ICW4 0x01
#define ICW4_x86 0x01

void cls();
void setMonitorColor(char);

void printString(char*);
void printChar(char);

void scroll();

void printColorString(char* , char);
void printColorChar(char , char);

void getDecAscii(int);

void initIDT();
extern  void loadIdt();
extern  void isr1_Handler();
void handleKeypress(int);
void pressed(char);
void picRemap();

unsigned char inportb(unsigned short);
void outportb(unsigned short , unsigned char);

char* TM_START;
char NumberAscii[10];
int CELL;

struct IDT_ENTRY{
    unsigned short base_Lower;
    unsigned short selector;
    unsigned char zero;
    unsigned char flags;
    unsigned short base_Higher;
};

struct IDT_ENTRY idt[256];
extern unsigned int isr1;
unsigned int base;

int start(){
    TM_START = (char*) 0xb8000;
    CELL = 0;
    base = (unsigned int)&isr1;
```

```
53
54      cls();
55      setMonitorColor(0xa5);
56
57      char Welcome[] = "Welcome To OS0 : Copyright 2021\n";
58      char Welcome2[] = "Command Line Version 1.0.0.0\n\n";
59      char OSM[] = "OS0 > ";
60
61      printString(Welcome);
62      printString(Welcome2);
63      printColorString(OSM , 0xa8);
64
65      initIDT();
66  }
67
68
69  void cls(){
70      int i = 0;
71      CELL = 0;
72      while(i < (2 * 80 * 25)){
73              *(TM_START + i) = ' '; // Clear screen
74              i += 2;
75      }
76  }
77
78  void setMonitorColor(char Color){
79      int i = 1;
80      while(i < (2 * 80 * 25)){
81              *(TM_START + i) = Color;
82              i += 2;
83      }
84  }
85
86  void printString(char* cA){
87      int i = 0;
88      while(*(cA + i) != '\0'){
89              printChar(*(cA + i));
90              i++;
91      }
92  }
93
94  void printChar(char c){
95      if(CELL == 2 * 80 * 25)
96              scroll();
97      if(c == '\n'){
98              CELL = ((CELL + 160) - (CELL % 160));
99              return;
100     }
101     *(TM_START + CELL) = c;
102     CELL += 2;
103 }
104
105 void scroll(){
106     int i = 160 , y = 0;
107     while(i < 2 * 80 * 25){
108             *(TM_START + y) = *(TM_START + i);
109             i += 2;
```

```
110             y += 2;
111     }
112     CELL = 2 * 80 * 24;
113     i = 0;
114     while(i < 160){
115             *(TM_START + CELL + i) = ' ';
116             i += 2;
117     }
118 }
119
120 void printColorString(char* c , char co){
121     int i = 0;
122     while(*(c + i) != '\0'){
123             printColorChar(*(c + i) , co);
124             i++;
125     }
126 }
127
128 void printColorChar(char c , char co){
129     if(CELL == 2 * 80 * 25)
130             scroll();
131     if(c == '\n'){
132             CELL = ((CELL + 160) - (CELL % 160));
133             return;
134     }
135     *(TM_START + CELL) = c;
136     *(TM_START + CELL + 1) = co;
137     CELL += 2;
138 }
139
140 void getDecAscii(int num){
141     if(num == 0){
142             NumberAscii[0] = '0';
143             return;
144     }
145     char NUM[10];
146     int i = 0 , j = 0;
147     while(num > 0){
148             NUM[i] = num % 10;
149             num /= 10;
150             i++;
151     }
152     i--;
153     while(i >= 0){
154             NumberAscii[j] = NUM[i];
155             i--;
156             j++;
157     }
158     NumberAscii[j] = 'J';
159     j = 0;
160     while(NumberAscii[j] != 'J'){
161             NumberAscii[j] = '0' + NumberAscii[j];
162             j++;
163     }
164     NumberAscii[j] = 0;
165 }
166
```

```
167 void initIDT(){
168     idt[1].base_Lower = (base & 0xFFFF);
169     idt[1].base_Higher = (base >> 16) & 0xFFFF;
170     idt[1].selector = 0x08;
171     idt[1].zero = 0;
172     idt[1].flags = 0x8e;
173 
174     picRemap();
175 
176     outportb(0x21 , 0xfd);
177     outportb(0xa1 , 0xff);
178 
179     loadIdt();
180 }
181 
182 unsigned char inportb(unsigned short _port){
183     unsigned char rv;
184     __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
185     return rv;
186 }
187 
188 void outportb(unsigned short _port, unsigned char _data){
189     __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
190 }
191 
192 extern void isr1_Handler(){
193     handleKeypress(inportb(0x60));
194     outportb(0x20 , 0x20);
195     outportb(0xa0 , 0x20);
196 }
197 
198 void handleKeypress(int code){
199     char Scancode[] = {
200             0 , 0 , '1' , '2' ,
201             '3' , '4' , '5' , '6' ,
202             '7' , '8' , '9' , '0' ,
203             '-' , '=' , 0 , 0 , 'Q' ,
204             'W' , 'E' , 'R' , 'T' , 'Y' ,
205             'U' , 'I' , 'O' , 'P' , '[' , ']' ,
206             0 , 0 , 'A' , 'S' , 'D' , 'F' , 'G' ,
207             'H' , 'J' , 'K' , 'L' , ';' , '\'' , '`' ,
208             0 , '\\' , 'Z' , 'X' , 'C' , 'V' , 'B' , 'N' , 'M' ,
209             ',' , '.' , '/' , 0 , '*' , 0 , ' '
210     };
211 
212     if(code == 0x1c)
213             printChar('\n');
214     else if(code < 0x3a)
215             pressed(Scancode[code]);
216 }
217 
218 void pressed(char key){
219     printChar(key);
220 }
221 
222 void picRemap(){
223     unsigned char a , b;
```

```
224     a = inportb(PIC1_D);
225     b = inportb(PIC2_D);
226 
227     outportb(PIC1_C , ICW1_DEF | ICW1_ICW4);
228     outportb(PIC2_C , ICW1_DEF | ICW1_ICW4);
229 
230     outportb(PIC1_D , 0);
231     outportb(PIC2_D , 8);
232 
233     outportb(PIC1_D , 4);
234     outportb(PIC2_D , 2);
235 
236     outportb(PIC1_D , ICW4_x86);
237     outportb(PIC2_D , ICW4_x86);
238 
239     outportb(PIC1_D , a);
240     outportb(PIC2_D , b);
241 }
```

Kernel_Entry.asm

```
1 START:
2 [bits 32]
3 [extern _start]
4       call _start
5       jmp $
```

IDT.asm

```
 1 extern _idt
 2 extern _isr1_Handler
 3 global _isr1
 4 global _loadIdt
 5 
 6 idtDesc:
 7       dw 2048
 8       dd _idt
 9 
10 _isr1:
11       pushad
12       call _isr1_Handler
13       popad
14       iretd
15 
16 _loadIdt:
17       lidt[idtDesc]
18       sti
19       ret
```

compile.bat

```
nasm Boot.asm -f bin -o bin\bootsect.bin
nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
nasm IDT.asm -f elf -o bin\IDT.bin

gcc -m32 -ffreestanding -c main.c -o bin\kernel.o

ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
bin\IDT.bin

objcopy -O binary -j .text  bin\kernel.img bin\kernel.bin
copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image

qemu-system-i386 -drive format=raw,file=bin\os-image
```

Boot.asm

```
[org 0x7c00]
[bits 16]

;Boot Loader
mov bx , 0x1000 ; Memory offset to which kernel will be loaded
mov ah , 0x02   ; Bios Read Sector Function
mov al , 30     ; No. of sectors to read(If your kernel won't fit into 30
sectors , \
you may need to provide the correct no. of sectors to read)
mov ch , 0x00   ; Select Cylinder 0 from harddisk
mov dh , 0x00   ; Select head 0 from hard disk
mov cl , 0x02   ; Start Reading from Second sector(Sector just after boot
sector)

int 0x13        ; Bios Interrupt Relating to Disk functions


;Switch To Protected Mode
cli ; Turns Interrupts off
lgdt [GDT_DESC] ; Loads Our GDT

mov eax , cr0
or  eax , 0x1
mov cr0 , eax ; Switch To Protected Mode

jmp  CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
;Forces the cpu to flush out contents in cache memory

[bits 32]

INIT_PM:
mov ax , DATA_SEG
mov ds , ax
mov ss , ax
mov es , ax
mov fs , ax
mov gs , ax

mov ebp , 0x90000
```

```
38 mov esp , ebp ; Updates Stack Segment
39
40
41 call 0x1000
42 jmp $
43
44
45
46
47
48 GDT_BEGIN:
49
50 GDT_NULL_DESC:  ;The  Mandatory  Null  Descriptor
51      dd 0x0
52      dd 0x0
53
54 GDT_CODE_SEG:
55      dw 0xffff               ;Limit
56      dw 0x0                  ;Base
57      db 0x0                  ;Base
58      db 10011010b    ;Flags
59      db 11001111b    ;Flags
60      db 0x0                  ;Base
61
62 GDT_DATA_SEG:
63      dw 0xffff               ;Limit
64      dw 0x0                  ;Base
65      db 0x0                  ;Base
66      db 10010010b    ;Flags
67      db 11001111b    ;Flags
68      db 0x0                  ;Base
69
70 GDT_END:
71
72 GDT_DESC:
73      dw GDT_END - GDT_BEGIN - 1
74      dd GDT_BEGIN
75
76 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
77 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
78
79
80 times 510-($-$$) db 0
81 dw 0xaa55
```

Also note to create a folder named `bin` in the same directory.

Lets look at how this works:

In the `main.c` file, in the `start()` function, Look at the last line: We called the `initIDT()` function.

The job of this function is to set the Interrupt Descriptor Table so that when a key is pressed, It will call the `isr1_Handler()` function.

Look at the very top of this file;

```
struct IDT_ENTRY{

    unsigned short base_Lower;
    unsigned short selector;
    unsigned char zero;
    unsigned char flags;
    unsigned short base_Higher;

};

struct IDT_ENTRY idt[256];
extern unsigned int isr1;
unsigned int base;
```

The struct `IDT_ENTRY` could be used to define the `idt`. There are a total of `256` possible idt entries. So we created 256 of them with `struct IDT_ENTRY idt[256];`

The next command `extern unsigned int isr1;` says to the compiler to implement compilation so that it knows that there will be a section named `isr1` in an outside source file. We could find this section in the `IDT.asm` file.

In the `start()` function in `main.c`, we copied the address of the `isr1` section to the variable `base` with `base = (unsigned int)&isr1;` (We used the & Symbol to get the address).

In the `initIDT()` function, we mapped the #1 interrupt(Keyboard interrupt(IRQ 1)) with:

```
    idt[1].base_Lower = (base & 0xFFFF);
```

```
    idt[1].base_Higher = (base >> 16) & 0xFFFF;
    idt[1].selector = 0x08;
    idt[1].zero = 0;

    idt[1].flags = 0x8e;
```

Here we set the location of `isr1` section in the first two lines using the value in `base` variable.

The next three variables sets the arguments necessary for the idt entry.

Then we prepared the pic chip to handle interrupts and finally, we called the `loadIdt()` function. At the very top of the code, we said to the compiler that the `loadIdt()` function wil be outside the c source file with `extern void loadIdt();`

When the `loadIdt()` code is executed, it jumps to that section we defined in the `IDT.asm` file.

```
_loadIdt:


    lidt[idtDesc]
    sti
    ret
```

`lidt` is a x86 command used to make the processor load the Interrupt Descriptor Table. Here, it loads the idt entry Which the `idtDesc` section points to.

The command `sti` is used to enable the interrupts. From now onwards , we will get every interrupts.

From now onwards, the `_isr1` section in IDT.asm file will be called when ever a keyboard key is being pressed.

The `_isr1` section calls the `_isr1_Handler` function we defined in the `main.c` file.

When ever a key is pressed, the fuction `isr1_Handler()` will be called. We could implement what ever to do in that function.

According to our implementation we first obtained the scan code of key being pressed with `inportb(0x60)` and passed it to the `handleKeypress()` function with `handleKeypress(inportb(0x60));`

In the `handleKeypress()` function, we have an array of values named `Scancode[]`.
We could take the scan code as an index to point to the `Scancode[]` array.

The arrangement of values in `Scancode[]` gives us the ascii representation of the scan code.

Finally we called a function to print the character being pressed.

```
THE inportb AND outportb FUNCTIONS ARE FUNCTIONS THAT HELP US
COMMUNICATE WITH EXTERNAL DEVICES. in COMMAND ACCEPTS INPUT
FROM EXTERNAL DEVICES AND out COMMAND OUTPUTS COMMANDS AND
DATA TO EXTERNAL DEVICES.
```

Compile and run the code by executing the `compile.bat` file and try pressing any character on keyboard, The program simply prints it to the screen.

External References

- https://wiki.osdev.org/8259_PIC
- https://wiki.osdev.org/Interrupts
- https://www.osdev.org/howtos/2/
- https://wiki.osdev.org/Interrupt_Descriptor_Table

Making Our First Prototype : OS0

Introduction

In this section, We will try to implement everything we have learned this far. We will make a program which changes the colour of screen depending on the command it receives , play videos , print string etc…. This chapter just gives you an idea about practical implementation.

Developing the First Prototype

Use the following link to get the full source code for this section:

`https://github.com/TINU-2000/OS-DEV/tree/main/First%20Prototype`

`main.c`

```c
#include "extra.h"

#define PIC1_C 0x20
#define PIC1_D 0x21
#define PIC2_C 0xa0
#define PIC2_D 0xa1

#define ICW1_DEF 0x10
#define ICW1_ICW4 0x01
#define ICW4_x86 0x01

void cls();
void setMonitorColor(char);

void printString(char*);
void printChar(char);

void scroll();

void printColorString(char* , char);
void printColorChar(char , char);

void getDecAscii(int);

void initIDT();
extern  void loadIdt();
```

```
27 extern  void isr1_Handler();
28 void handleKeypress(int);
29 void pressed(char);
30 void picRemap();
31
32 unsigned char inportb(unsigned short);
33 void outportb(unsigned short , unsigned char);
34
35
36 char NumberAscii[10];
37 int CELL;
38
39
40 char COMMAND[21];
41 int i = 0;
42
43 struct IDT_ENTRY{
44     unsigned short base_Lower;
45     unsigned short selector;
46     unsigned char zero;
47     unsigned char flags;
48     unsigned short base_Higher;
49 };
50
51 struct IDT_ENTRY idt[256];
52 extern unsigned int isr1;
53 unsigned int base;
54
55 int start(){
56     TM_START = (char*) 0xb8000;
57     CELL = 0;
58     base = (unsigned int)&isr1;
59
60     cls();
61     setMonitorColor(0xa5);
62
63     char Welcome[] = "Welcome To OS0 : Copyright 2021\n";
64     char Welcome2[] = "Command Line Version 1.0.0.0\n\n";
65     char OSM[] = "OS0 > ";
66
67     printString(Welcome);
68     printString(Welcome2);
69     printColorString(OSM , 0xa8);
70
71     initIDT();
72 }
73
74
75 void cls(){
76     int i = 0;
77     CELL = 0;
78     while(i < (2 * 80 * 25)){
79             *(TM_START + i) = ' '; // Clear screen
80             i += 2;
81     }
82 }
83
```

```
84 void setMonitorColor(char Color){
85     int i = 1;
86     while(i < (2 * 80 * 25)){
87             *(TM_START + i) = Color;
88             i += 2;
89     }
90 }
91
92 void printString(char* cA){
93     int i = 0;
94     while(*(cA + i) != '\0'){
95             printChar(*(cA + i));
96             i++;
97     }
98 }
99
100 void printChar(char c){
101     if(CELL == 2 * 80 * 25)
102             scroll();
103     if(c == '\n'){
104             CELL = ((CELL + 160) - (CELL % 160));
105             return;
106     }
107     *(TM_START + CELL) = c;
108     CELL += 2;
109 }
110
111 void scroll(){
112     int i = 160 , y = 0;
113     while(i < 2 * 80 * 25){
114             *(TM_START + y) = *(TM_START + i);
115             i += 2;
116             y += 2;
117     }
118     CELL = 2 * 80 * 24;
119     i = 0;
120     while(i < 160){
121             *(TM_START + CELL + i) = ' ';
122             i += 2;
123     }
124 }
125
126 void printColorString(char* c , char co){
127     int i = 0;
128     while(*(c + i) != '\0'){
129             printColorChar(*(c + i) , co);
130             i++;
131     }
132 }
133
134 void printColorChar(char c , char co){
135     if(CELL == 2 * 80 * 25)
136             scroll();
137     if(c == '\n'){
138             CELL = ((CELL + 160) - (CELL % 160));
139             return;
```

```
140     }
141     *(TM_START + CELL) = c;
142     *(TM_START + CELL + 1) = co;
143     CELL += 2;
144 }
145
146 void getDecAscii(int num){
147     if(num == 0){
148             NumberAscii[0] = '0';
149             return;
150     }
151     char NUM[10];
152     int i = 0 , j = 0;
153     while(num > 0){
154             NUM[i] = num % 10;
155             num /= 10;
156             i++;
157     }
158     i--;
159     while(i >= 0){
160             NumberAscii[j] = NUM[i];
161             i--;
162             j++;
163     }
164     NumberAscii[j] = 'J';
165     j = 0;
166     while(NumberAscii[j] != 'J'){
167             NumberAscii[j] = '0' + NumberAscii[j];
168             j++;
169     }
170     NumberAscii[j] = 0;
171 }
172
173 void initIDT(){
174     idt[1].base_Lower = (base & 0xFFFF);
175     idt[1].base_Higher = (base >> 16) & 0xFFFF;
176     idt[1].selector = 0x08;
177     idt[1].zero = 0;
178     idt[1].flags = 0x8e;
179
180     picRemap();
181
182     outportb(0x21 , 0xfd);
183     outportb(0xa1 , 0xff);
184
185     loadIdt();
186 }
187
188 unsigned char inportb(unsigned short _port){
189     unsigned char rv;
190     __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
191     return rv;
192 }
193
194 void outportb(unsigned short _port, unsigned char _data){
195     __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
196 }
```

```
197
198 extern void isr1_Handler(){
199     handleKeypress(inportb(0x60));
200     outportb(0x20 , 0x20);
201     outportb(0xa0 , 0x20);
202 }
203
204 void handleKeypress(int code){
205     char OSM[] = "\nOS0 > ";
206     char Scancode[] = {
207             0 , 0 , '1' , '2' ,
208             '3' , '4' , '5' , '6' ,
209             '7' , '8' , '9' , '0' ,
210             '-' , '=' , 0 , 0 , 'Q' ,
211             'W' , 'E' , 'R' , 'T' , 'Y' ,
212             'U' , 'I' , 'O' , 'P' , '[' , ']' ,
213             0 , 0 , 'A' , 'S' , 'D' , 'F' , 'G' ,
214             'H' , 'J' , 'K' , 'L' , ';' , '\'' , '`' ,
215             0 , '\\' , 'Z' , 'X' , 'C' , 'V' , 'B' , 'N' , 'M' ,
216             ',' , '.' , '/' , 0 , '*' , 0 , ' '
217     };
218
219     if(code == 0x1c){
220             COMMAND[i] = '\0';
221             i = 0;
222             strEval(COMMAND);
223             printString(OSM);
224     }
225     else if(code < 0x3a)
226             pressed(Scancode[code]);
227 }
228
229 void pressed(char key){
230     if(i != 20){
231             COMMAND[i] = key;
232             i++;
233             printChar(key);
234     }
235     else{
236             blink();
237     }
238 }
239
240 void picRemap(){
241     unsigned char a , b;
242     a = inportb(PIC1_D);
243     b = inportb(PIC2_D);
244
245     outportb(PIC1_C , ICW1_DEF | ICW1_ICW4);
246     outportb(PIC2_C , ICW1_DEF | ICW1_ICW4);
247
248     outportb(PIC1_D , 0);
249     outportb(PIC2_D , 8);
250
251     outportb(PIC1_D , 4);
252     outportb(PIC2_D , 2);
253
```

```
254     outportb(PIC1_D , ICW4_x86);
255     outportb(PIC2_D , ICW4_x86);
256
257     outportb(PIC1_D , a);
258     outportb(PIC2_D , b);
259 }
```

extra.h

```
 1 void setMonitorColor(char);
 2 void cls();
 3 void printString(char*);
 4 void vid();
 5
 6 char* TM_START;
 7
 8 void blink(){
 9     setMonitorColor(0x59);
10     int TIME_OUT = 0x10fffff;
11     while(--TIME_OUT);
12     setMonitorColor(0xa5);
13 }
14
15 char strcmp(char* sou , char* dest){
16     int i = 0;
17     while(*(sou + i) == *(dest + i)){
18             if(*(sou + i) == 0 && *(dest + i) == 0)
19                     return 1;
20             i++;
21     }
22     return 0;
23 }
24
25 void strEval(char* CMD){
26     char cmd1[] = "CLS";
27     char cmd2[] = "COLORA";
28     char cmd3[] = "COLORB";
29     char cmd4[] = "COLORC";
30     char cmd5[] = "COLORDEF";
31     char cmd6[] = "VID";
32     char cmd7[] = "HI";
33
34     char msg1[] = "\nHELLO , HAVE A GOOD JOURNEY LEARNING\n";
35
36     if(strcmp(CMD , cmd1))
37             cls();
38
39     else if(strcmp(CMD , cmd2))
40             setMonitorColor(0x3c);
41
42     else if(strcmp(CMD , cmd3))
43             setMonitorColor(0x5a);
44
45     else if(strcmp(CMD , cmd4))
46             setMonitorColor(0x2a);
```

```
47
48      else if(strcmp(CMD , cmd5))
49              setMonitorColor(0xa5);
50
51      else if(strcmp(CMD , cmd6))
52              vid();
53      else if(strcmp(CMD , cmd7))
54              printString(msg1);
55  }
56
57  void vid(){
58      char clr = 'A';
59      while(1){
60              int i = 0;
61              while(i < 2 * 80 * 25){
62                      *(TM_START + i) = clr;
63                      clr++;
64                      i++;
65              }
66      }
67  }
```

Explanation

We have implemented a small amount of application in this code. Here is the list:

After compiling and executing the code, We will be able to type to the screen. If the number of characters in a line becomes 20, Further key presses will generate a blinking in the screen. We did this by setting the colour of the screen for a small amount of time and changing it back to the orginal color.

hitting enter key will allow us to write to the next line.

Typing `HI` and hitting enter will display a pre defined text to the screen. Typing `COLORA` , `COLORB` , `COLORC` and `COLORDEF` will change the color of the screen.

Typing `CLS` will clear the screen.

Typing `VID` will generate a video feeling look. Here you need to relaunch the executable to do further experiments. This is because as the computer is fully working on video playing, it will not have time to exit from it when we say to it for eg: when pressing a key.

We could easily solve this problem by using `Threading`, But currently, we are not at that topic.

The function `strEval()` in `extra.h` is the function which evaluates the commands and performs necessary operation.

We made the call to this function from the function named `handleKeypress()` in `main.c` inside an `if` condition :

```
    if(code == 0x1c){

    }
```

This is so that `strEval()` will only be called when the enter key is pressed.

TRY MAKING YOUR OWN IMPLEMENTATION OF THIS PROGRAM.
OR YOU CAN TRY MAKING SOME 2D GAME WHICH YOU CAN CONTROL WITH THE KEYBOARD.
YOU WILL BE VERY CONFIDENT IN THIS TOPIC AFTER YOU INITIATE YOUR OWN PROJECTS.

In later chapters, we will discuss about Creating pixel by pixel drawing. But This `Text Mode` graphics can also be used to implement a gui as we have the option to draw coloumn based colors. Earlier computers even used text mode graphics capabilities to make a GUI.

Accessing Hard Disks

Introduction

`Hard Disk` or the Primary `"Secondary Storage Device"` is also one of the important part of a computer. Along with storage uses, it's abilities are also used to maximize the use of Primary storage device or RAM. Concepts such as paging make use of this.

When implementing a hard disk driver , we usually use the `in` and `out` assembly commands. Before developing a hard disk driver, we need to know the types of hard disks and how the it is organized.

Working With Hard Disks

Types of Hard Disk

The two main types of hard disk available today are `HDD` and `SSD`. HDD Stands for Hard Disk Drive and SSD stands for Solid State Drive.

HDD

Hard Disk Drive or HDD is a mechanical hard disk with a rotating platter and a moving head. Data is stored into it in magnetic form. The Platter(Where data is stored) , rotates during read or write operations and the Head(The part of hard disk which do the read/write operation) performs the desired operation on the platter.

Even though the RPM of modern hard disks are pretty huge, It is little slower when comparing with the speed of Main Memory or RAM, as every read or write operation needs the head to move to desired locations.

SSD

SSD or Solid State Drive is a non mechanical type of hard disk where the data is stored typically using flash memory technology. As it is non

mechanical , SSD's are much faster(But costly too). SSD's will give us a faster boot time , load time and even increase the process execution speed. This is because the paging operation will be much faster. Paging is a concept used to utilize main memory to its fullest by putting "less often used" data temporarily to hard Hard disk and later copy it back to memory when needed.

How Hard Disk is Divided

Let's now learn how the HDD is organized to store data.

We said that the data is stored in magnetic form in platters. Typically hard disks have one to four platters stacked together. All of this platters will have separate Heads to read/write data.

All of this platters are further divided into `tracks` and `sectors`. There are number of tracks in a hard disk. All of this tracks will have a set of sectors in it. Usually the size of one sector is `512` bytes.

Another term associated with the hard disk is `cylinder`.

We already said that hard disks have number of platters and each platters are further divided into tracks. A cylinder is formed by joining the same tracks in all of the platters. For eg: `Track 1` of `Platter 1` + `Track 1` of `Platter 2` + `Track 1` of `Platter 3` ..... Forms one cylinder.

Implementing a Hard Disk Driver

The full source code for this section is at:

`https://github.com/TINU-2000/OS-DEV/tree/main/Hard%20Disk%20-%20ATA`

`ata.asm`

```
1 [bits 32]
2
3 extern _blockAddr
4 extern _At
5 global _read
6 global _write
```

```
 7
 8 _read:
 9
10             pushfd
11             and eax , 0x0FFFFFFF
12             push eax
13             push ebx
14             push ecx
15             push edx
16             push edi
17
18             mov eax , [_blockAddr]
19             mov cl , 1
20             mov edi , _At
21
22             mov ebx , eax
23
24             mov edx , 0x01F6
25             shr eax , 24
26             or al , 11100000b     ; Sets bit 6 in al for LBA mode
27             out dx , al
28
29             mov edx , 0x01F2      ; Port to send number of sectors
30             mov al , cl           ; Get number of sectors from CL
31             out dx , al
32
33             mov edx , 0x1F3       ; Port to send bit 0 - 7 of LBA
34             mov eax , ebx         ; Get LBA from EBX
35             out dx , al
36
37             mov edx , 0x1F4       ; Port to send bit 8 - 15 of LBA
38             mov eax , ebx         ; Get LBA from EBX
39             shr eax , 8           ; Get bit 8 - 15 in AL
40             out dx , al
41
42
43             mov edx , 0x1F5       ; Port to send bit 16 - 23 of LBA
44             mov eax , ebx         ; Get LBA from EBX
45             shr eax , 16          ; Get bit 16 - 23 in AL
46             out dx , al
47
48             mov edx , 0x1F7       ; Command port
49             mov al , 0x20         ; Read with retry.
50             out dx , al
51
52 .loop1:
53             in al , dx
54             test al , 8
55             jz .loop1
56
57             mov eax , 256         ; Read 256 words , 1 sector
58             xor bx , bx
59             mov bl , cl           ; read CL sectors
60             mul bx
61             mov ecx , eax
62             mov edx , 0x1F0
63             rep insw             ; copy to [RDI]
```

```
64
65              pop edi
66              pop edx
67              pop ecx
68              pop ebx
69              pop eax
70              popfd
71              ret
72
73
74  _write:
75              pushfd
76              and eax , 0x0FFFFFFF
77              push eax
78              push ebx
79              push ecx
80              push edx
81              push edi
82
83              mov eax , [_blockAddr]
84              mov cl , 1
85              mov edi , _At
86
87              mov ebx , eax
88
89              mov edx , 0x01F6
90              shr eax , 24
91              or al , 11100000b     ; Set bit 6 in al for LBA mode
92              out dx , al
93
94              mov edx , 0x01F2      ; Port to send number of sectors
95              mov al , cl           ; Get number of sectors from CL
96              out dx , al
97
98              mov edx , 0x1F3       ; Port to send bit 0 - 7 of LBA
99              mov eax , ebx         ; Get LBA from EBX
100             out dx , al
101
102             mov edx , 0x1F4       ; Port to send bit 8 - 15 of LBA
103             mov eax , ebx         ; Get LBA from EBX
104             shr eax , 8           ; Get bit 8 - 15 in AL
105             out dx , al
106
107
108             mov edx , 0x1F5       ; Port to send bit 16 - 23 of LBA
109             mov eax , ebx         ; Get LBA from EBX
110             shr eax , 16          ; Get bit 16 - 23 in AL
111             out dx , al
112
113             mov edx , 0x1F7       ; Command port
114             mov al , 0x30         ; Write with retry.
115             out dx , al
116
117 .loop2:
118             in al , dx
119             test al , 8
120             jz .loop2
```

```
121
122             mov eax , 256         ; Read 256 words , 1 sector
123             xor bx , bx
124             mov bl , cl           ; write CL sectors
125             mul bx
126             mov ecx , eax
127             mov edx , 0x1F0
128             mov esi , edi
129             rep outsw            ; go out
130
131             pop edi
132             pop edx
133             pop ecx
134             pop ebx
135             pop eax
136             popfd
137             ret
```

extra.h

```
 1 void setMonitorColor(char);
 2 void cls();
 3 void printString(char*);
 4 void vid();
 5 void put();
 6 void get();
 7
 8 extern void read();
 9 extern void write();
10
11
12 int blockAddr;
13 char At[1024];
14
15 char* TM_START;
16
17 void blink(){
18     setMonitorColor(0x59);
19     int TIME_OUT = 0x10fffff;
20     while(--TIME_OUT);
21     setMonitorColor(0xa5);
22 }
23
24 char strcmp(char* sou , char* dest){
25     int i = 0;
26     while(*(sou + i) == *(dest + i)){
27             if(*(sou + i) == 0 && *(dest + i) == 0)
28                     return 1;
29             i++;
30     }
31     return 0;
32 }
33
34 void strEval(char* CMD){
35     char cmd1[] = "CLS";
```

```
    char cmd2[] = "COLORA";
    char cmd3[] = "COLORB";
    char cmd4[] = "COLORC";
    char cmd5[] = "COLORDEF";
    char cmd6[] = "VID";
    char cmd7[] = "HI";
    char cmd8[] = "PUT";
    char cmd9[] = "GET";

    char msg1[] = "\nHELLO , HAVE A GOOD JOURNEY LEARNING\n";

    if(strcmp(CMD , cmd1))
            cls();

    else if(strcmp(CMD , cmd2))
            setMonitorColor(0x3c);

    else if(strcmp(CMD , cmd3))
            setMonitorColor(0x5a);

    else if(strcmp(CMD , cmd4))
            setMonitorColor(0x2a);

    else if(strcmp(CMD , cmd5))
            setMonitorColor(0xa5);

    else if(strcmp(CMD , cmd6))
            vid();
    else if(strcmp(CMD , cmd7))
            printString(msg1);
    else if(strcmp(CMD , cmd8)){
            blockAddr = 0;
            int i = 0;

            while(i < 511){
                    At[i] = 'J'; // Fill with J
                    i++;
            }
            At[i] = 0; // Null character

            put(); // Writes to Hard disk

            i = 0;
            while(i < 511){
                    At[i] = 0;  // Clears the content
                    i++;
            }
    }
    else if(strcmp(CMD , cmd9)){
            blockAddr = 0;
            get();
            printString(At);
    }

}

void vid(){
```

```
 93     char clr = 'A';
 94     while(1){
 95             int i = 0;
 96             while(i < 2 * 80 * 25){
 97                     *(TM_START + i) = clr;
 98                     clr++;
 99                     i++;
100             }
101     }
102 }
103
104 void put(){
105     write();
106 }
107
108 void get(){
109     read();
110 }
```

compile.bat

```
 1 nasm Boot.asm -f bin -o bin\bootsect.bin
 2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
 3 nasm IDT.asm -f elf -o bin\IDT.bin
 4 nasm ata.asm -f elf -o bin\ata.bin
 5
 6 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
 7
 8 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
bin\IDT.bin bin\
 9 \ata.bin
10
11 objcopy -O binary -j .text  bin\kernel.img bin\kernel.bin
12 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
13
14 qemu-system-i386 -drive format=raw,file=bin\os-image
```

How it Works?

Here, we implemented a hard disk driver for the ATA technology. ATA or Advanced Technology Attachment allows hard disks and CD-ROMs to be internally connected to the motherboard and perform input/output functions.

There are different ways to read/write to hard disk. What we have used is LBA mode.

This is the easiest way to read/write to hard disk , all we need to do is pass the Block address of sector. Passing 0 will give us access to the first

sector(Boot sector).

Please note not to write to the 0'th sector as this can make your computer non bootable, but you could always copy a boot loader to that sector.

Let's now understand the code:

The two new commands we added to the `strEval` function are `GET` and `PUT`. When we type and enter the `PUT` command, it first copies the number `0` to `blockAddr` and then proceeds to initialize every cell in `At` character array with `'J'` and finally adds a null character. Then it calls the `put` function which calls another function named `write`.

The `write` function is defined in `ata.asm` file. In that function, look at the following section:

```
mov eax , [_blockAddr]

mov cl , 1
mov edi , _At
```

Here , the program copies the value we copied to the `blockAddr` variable in `strEval` function to the eax register. This value represents which sector to write to hard disk. Then we copied the value `1` to `cl` register , this represents the number of sectors to write. As we give `1` to that register, it writes 1 sector(512 bytes). After that, we copied the address of the array `At` we defined in `extra.h` file. This is so that the processor will write the data in that array to hard disk. The rest of the code communicates with the hard disk to write to it, and after the operation ends, it returns to our code in `C`.

The same thing happens when we give the `GET` command. But instead of the writing process , it does the reading process. In the code , we first copied the value `0` to `blockAddr` variable, Then we called the `get` function which calls the `read` function in `ata.asm` file. That function reads the `0`'th sector to the `At` array and returns to the code in `C`. and finally prints the content in that array.

Now , when running it, try giving the `GET` command first. We will get some random characters as output.

Now , try giving the `PUT` command and later call the `GET` command. This will print a lot a `J` to the screen.

This is because , when we give the `PUT` command , it copies the string of `J`'s to `At` array and writes it to the hard disk. After that , when we call the `GET` command , it reads from the hard disk and outputs the string of `J`'s we previously copied to it.

We done all of these operations to the `0`'th sector. We could change the value in `blockAddr` variable to read/write to other sectors.

Now, as we developed the Hard disk driver, lets create a File System so that we could do some file operations.

External References

```
https://wiki.osdev.org/Category:ATA
https://wiki.osdev.org/ATA_PIO_Mode
```

Creating a Simple File System

Introduction

A `File System` is a system which lets us use the secondary storage device to store files , edit , rename , delete and do all other file operations. An ideal file system should be able to use the capacity of hard disk to the most. Fat , ntfs , ext etc… are examples of file systems.

We need to work on a good implementation so that it enables all of the file operations. The file system should be ideal enough so that even a single byte shouldn't be unused.

```
This is technically impossible to make use of every byte, But what it mean is that We should be able to use our hard disk to the most
```

Here , we will discuss how to implement a simple file system.

File system implementation is an advanced concept as it involves use of Data Structures.

We will discuss about implementing a simple file system which will be able to only save and retrieve some data. It won't be able to delete , rename or do other complicated stuff.

This is because as this is a beginner level book , we don't want the readers to be overwhelmed with big details.

The Implementation

Formatting

We are going to build a file system which allows us to give names to a file and store upto 512 byte data in that file.

Heres how we are going to implement it:

After our os is launched , before trying to create a file , The user needs to format the hard disk. This is usually done when the user enters the command `FORMAT` and presses enter key.

The formatting operation first stores four random bytes(Which the developer can choose) in the beginning of sector 0.

Then , the developer needs to code so that the remaining bytes in sector 0 and the rest of sectors initialize with 0. This is because , that we don't want the default data in hard disk to combine with the data we will store in it in the future. So it's a good idea to clear it. This is not so necessary but you may end up with problems depending upon your implementation.

```
Please note that the sector 0 is always used to store the
boot loader. Writing other data to sector 0 will make the
system un-bootable. if you are storing and booting the os
from real hardware , please choose other sector.
```

File Allocation Table And Storage Space

Create

The Sector 0 Where we stored the random bytes will be called as the file allocation table here. This sector could be used to store names of file that we will create in the future.

Now we will enable the users to type a command to create a file(You could choose any name for the command) and after obtaining the name for the file we do the following operations:

1. We check the first four bytes stored in sector 0 and if it is same as what we stored in the formatting section , we do the next operation.

```
We do this check to see if we have previously formatted the
disk. If the check fails , say to the user to enter the
command to format it.
```

2. Now we store the name of file in sector 0 just after the name of last file we created. Name of the very first file can be stored just after the first four bytes.

```
Please note that you need to manage the implementation so
that we could differentiate between different files
```

Save

To save some data to a file , the user first needs to create a file using the command you chose at the previous section. After creating the file , the user could attempt the `SAVE` operation we will discuss here.

Lets use the command `SAVE` to do the save operation. What the save operation does here is copying a specified string of text to hard disk.

After the user enters the save command , we will also obtain a file name and a string of text to save to hard disk. Now we will do a check to see whether the user previously created that file by looking to the 0'th sector and if it succeeds , we do the following operation

1. We obtains the index of the file specified. We will get 1 as index if the file specfied is the first file defined at sector 0. Like that , we will get 2 as index if the file specified is the second file defined at sector 0.
2. Now we take this index as the sector count where the string should be saved to.

if the index is 1 , we save the string of text to sector 1 , and if the index is 2 , we save the string to sector 2. and this process goes on.

```
Please note that we will only be able to store string of upto
512 bytes as the size of a sector is always 512 bytes.
Storing a large file needs different implementation.
```

I assume you already know how to access the sectors in hard disk. Heres how to do it (Assuming you already implemented the hard disk driver said in the previous chapter):

1. Store the index of sector to read/write to the variable `blockAddr` defined at `extra.h`. Copying 0 will give access to first sector(Boot sector) copying 1 will give access to 2'nd sector , copying 2 will give access to 3'rd sector……
2. Now to do the save operation , call the `put()` function and to retrieve the content from hard disk , call the `get()` function in `extra.h` header file.

Retrieve

To read the data in saved file , We will do things similar to what we done at `Save` Section. We first obtains the name of file , then searches the sector 0 to get index of that file and use that index to point to the sector and finally do the read operation.

Conclusion

The file system we discussed here is actually worth nothing. But i think it gives some ideas about how it should be done. An ideal file system should always support creating file of variable size , rename , delete , edit and other advanced things such as using algorithms to find best space to store a file. Finding best place to store a file is so crucial so that we could use the hard disk to the most.

Every file allocation methods have it's own pros and cons. But it's our responsibility to implement a system which best suit for our needs.

External References


```
https://wiki.osdev.org/File_Systems
https://wiki.osdev.org/FAT
```

Graphics Mode GUI Creation

Introduction

We this far have used Text Mode graphics to output to the screen. Even if it allows to print colours , we are not able to do it in a pixel by pixel manner. After switching to graphics mode , we will be able to draw pixel. We will now choose the screen resolution we want and do the video operations.

The Drawback of switching to Graphics mode is that , we need to design special programs to render text to the screen. We should first create a font and use it to render to screen whenever we want. Most Hobbyist operating systems use Text Mode graphics as it is the best solution to learn it. But in order to get the most of the systems abilities such as displaying icons , pictures etc… needs this switch.

Not all modes allow rendering full colours , we will choose from a list of modes which have different abilities.

Drawing In Graphics Mode

Modes

Choosing A Mode

When talking about switching to Graphics mode , we have different video modes to choose from. These different modes have its own abilities with its own supported resolution and colours.

The following table shows the supported modes and its representation hex value:
This is taken from:

`https://wiki.osdev.org/Drawing_In_Protected_Mode`

00 **text 40*25 16 color (mono)**
01 **text 40*25 16 color**
02 **text 80*25 16 color (mono)**
03 **text 80*25 16 color**
04 **CGA 320*200 4 color**
05 **CGA 320*200 4 color (m)**
06 **CGA 640*200 2 color**
07 **MDA monochrome text 80*25**
08 **PCjr**
09 **PCjr**
0A **PCjr**
0B **reserved**
0C **reserved**
0D **EGA 320*200 16 color**
0E **EGA 640*200 16 color**
0F **EGA 640*350 mono**
10 **EGA 640*350 16 color**
11 **VGA 640*480 mono**
12 **VGA 640*480 16 color**
13 **VGA 320*200 256 color**

We need to choose one from this list and note its associated hex value.
For now we will take **VGA 320*200 256 color** so its representation number hex value is 13.

Making Switch To The Selected Mode

We will now switch the video mode from our current Text Mode to Graphics Mode.
Switching to graphics mode is done using bios. But as we are in 32 bit Protected Mode , we won't be able to use bios. Bios is only available in 16 bit mode. So we will leave our current source codes for now and start a new project from scratch to show the development in graphics mode. After we switch to Graphics mode , we could switch to 32 bit protected mode.

```
Please note that we will use our current source codes in
later chapters. But for this chapter , we will create a
```

sample from scratch.

The following assembly statements switches to **VGA 320*200 256 color** Mode:

```
mov ah , 0x00
mov al , 0x13
int 0x10
```

Here , we first copied the value `0x00` to the `ah` register. This value represents the option in bios to switch video modes.

We then copied the value `0x13` to the `al` register, And finally called the bios with `int 0x10`.

Bios will now switch to **VGA 320*200 256 color** mode which we represented with `0x13`

We Could place this assembly statements in `Boot.asm` file just after the the Boot loader code.

We wont be able to switch to graphics mode in 32 bit protected mode. So the best place to put code to switch to Graphics mode in our case is just after the Boot Loader in `Boot.asm` file.

Please obtain the source code from:
`https://github.com/TINU-2000/OS-DEV/tree/main/Graphics%20Mode0`

`Boot.asm`

```
[org 0x7c00]
[bits 16]

;Boot Loader
mov bx , 0x1000 ; Memory offset to which kernel will be loaded
mov ah , 0x02   ; Bios Read Sector Function
mov al , 30     ; No. of sectors to read(If your kernel won't fit into 30 sectors , \
you may need to provide the correct no. of sectors to read)
mov ch , 0x00   ; Select Cylinder 0 from harddisk
```

```
10 mov dh , 0x00   ; Select head 0 from hard disk
11 mov cl , 0x02   ; Start Reading from Second sector(Sector just after boot
sector)
12
13 int 0x13        ; Bios Interrupt Relating to Disk functions
14
15 ;Switch to Graphics Mode
16 mov ah , 0x00
17 mov al , 0x13
18 int 0x10
19
20 ;Switch To Protected Mode
21 cli ; Turns Interrupts off
22 lgdt [GDT_DESC] ; Loads Our GDT
23
24 mov eax , cr0
25 or  eax , 0x1
26 mov cr0 , eax ; Switch To Protected Mode
27
28 jmp  CODE_SEG:INIT_PM ; Jumps To Our 32 bit Code
29 ;Forces the cpu to flush out contents in cache memory
30
31 [bits 32]
32
33 INIT_PM:
34 mov ax , DATA_SEG
35 mov ds , ax
36 mov ss , ax
37 mov es , ax
38 mov fs , ax
39 mov gs , ax
40
41 mov ebp , 0x90000
42 mov esp , ebp ; Updates Stack Segment
43
44
45 call 0x1000
46 jmp $
47
48
49
50
51
52 GDT_BEGIN:
53
54 GDT_NULL_DESC:  ;The  Mandatory  Null  Descriptor
55      dd 0x0
56      dd 0x0
57
58 GDT_CODE_SEG:
59      dw 0xffff               ;Limit
60      dw 0x0                  ;Base
61      db 0x0                  ;Base
62      db 10011010b    ;Flags
63      db 11001111b    ;Flags
64      db 0x0                  ;Base
65
```

```
66 GDT_DATA_SEG:
67      dw 0xffff               ;Limit
68      dw 0x0                  ;Base
69      db 0x0                  ;Base
70      db 10010010b    ;Flags
71      db 11001111b    ;Flags
72      db 0x0                  ;Base
73
74 GDT_END:
75
76 GDT_DESC:
77      dw GDT_END - GDT_BEGIN - 1
78      dd GDT_BEGIN
79
80 CODE_SEG equ GDT_CODE_SEG - GDT_BEGIN
81 DATA_SEG equ GDT_DATA_SEG - GDT_BEGIN
82
83
84 times 510-($-$$) db 0
85 dw 0xaa55
```

Kernel_Entry.asm

```
1 START:
2 [bits 32]
3 [extern _start]
4       call _start
5       jmp $
```

main.c

```
1 int start(){
2
3 }
```

compile.bat

```
1  nasm Boot.asm -f bin -o bin\bootsect.bin
2  nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3
4  gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
5
6  ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
7
8  objcopy -O binary -j .text  bin\kernel.img bin\kernel.bin
9  copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
10
11 qemu-system-i386 -drive format=raw,file=bin\os-image
```

Please note to create a folder named `bin` in the same folder as the `compile.bat` file saves the compiled files to it.

Look at the `Boot.asm` file to see changes.
Compile and run the program and you will get a black screen with no content. This is the desired output at this stage. We will now see how to draw to that screen.

Please also note that the `start()` function in `main.c` will be blank , we will now create code to render to the screen in that function.

Video Memory and Drawing

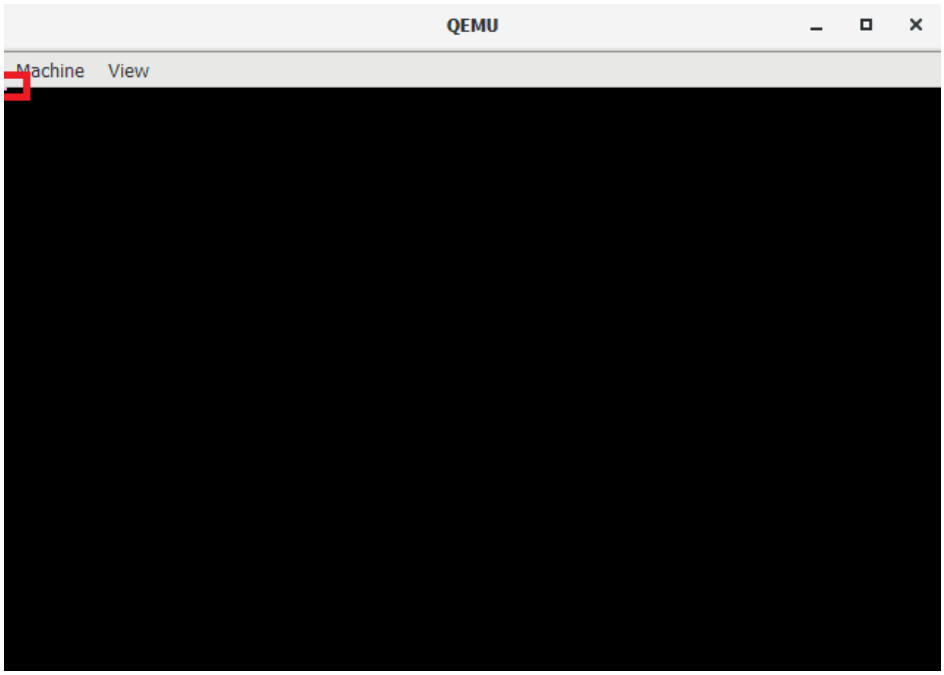
In Text Mode graphics , we wrote to a special location in ram to make things display to the screen. Graphics Mode Graphical operations also make use of memory to draw to the screen. But in this case , we will use `0xA0000` as address.

The following program renders one pixel to the Top Left of the screen by poking the very first video memory buffer byte:

`main.c`

```
1 int start(){
2       char* vbuff = (char*) 0xA0000;
3       *(vbuff) = 0x55;
4 }
```

Output



The pixel displayed here represents the hex value `0x55` which we copied to the video memory.

The supported values that can be copied to one cell is from `0x00` to `0xFF` or `0` to `255`

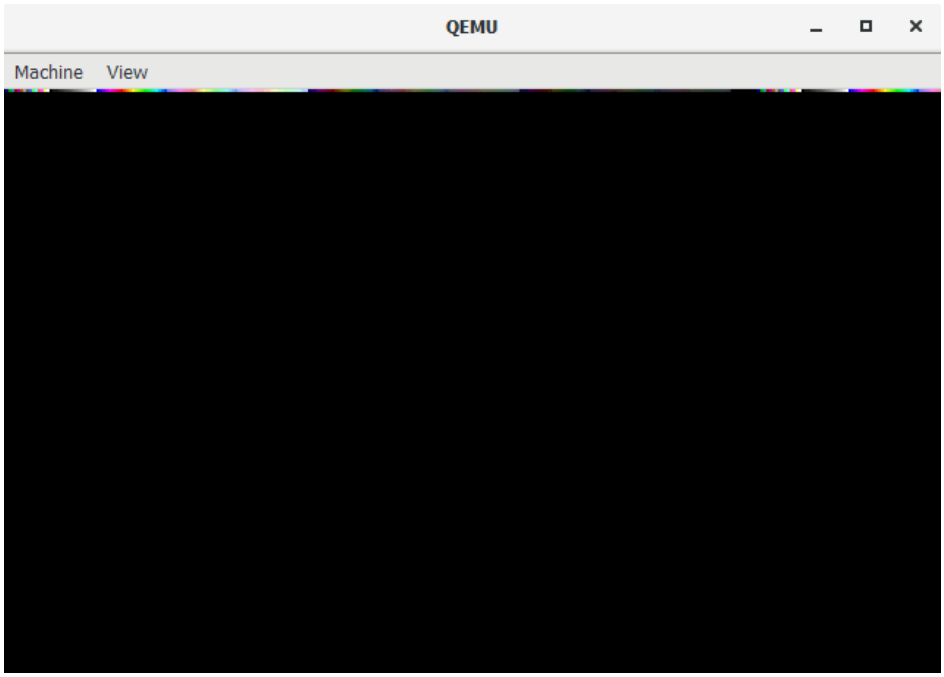
The rest of the pixels can be rendered by poking the rest of the video memory.
For eg: poking `0xA0000 + 1` renders the second pixel , poking `0xA0000 + 2` renders third pixel and this process goes on.

Look at the following program to poke every cell in first row:

`main.c`

```
int start(){
      char* vbuff = (char*) 0xA0000;
      char colr = 0x00;
      int i = 0;
      while(i < 320){
              *(vbuff + i) = colr;
              colr++;
              i++;
      }
}
```

Output



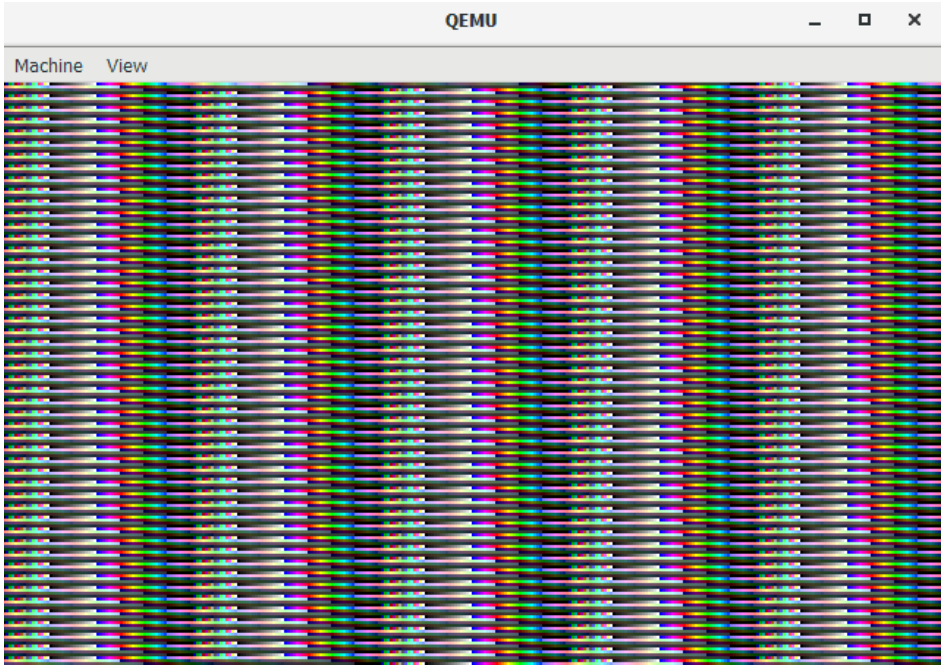
Now lets fill the screen:

`main.c`

```
int start(){
      char* vbuff = (char*) 0xA0000;
      char colr = 0x00;
      int i = 0;
      while(i < 320 * 200){
              *(vbuff + i) = colr;
              colr++;
              i++;
```

```
 9       }
10 }
```

Output



Sample User Interface Using Graphics Mode

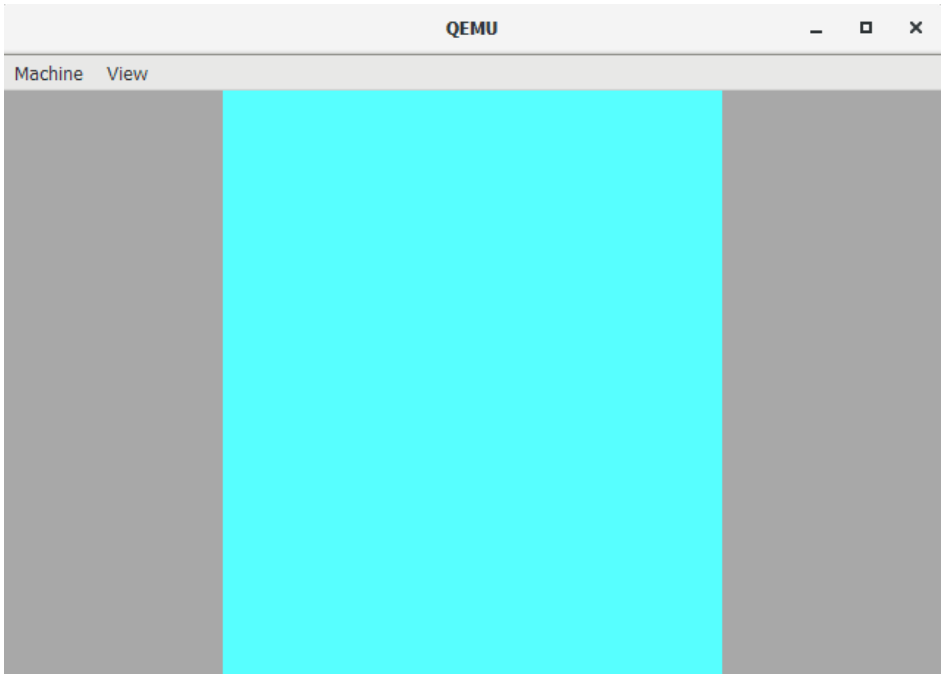
We have discussed about the base of creating a pixel by pixel drawing. Creating a good user interface needs deep knowledge in different colours and its values. Let me give a sample user interface , for that i will create a user interface looking like a text editor:

`main.c`

```
 1 int start(){
 2       char* vbuff = (char*) 0xA0000;
 3       int y = 0;
 4       while(y < 200){
 5               int x = 0;
 6               while(x < 75){
 7                       *(vbuff + (y * 320) + x) = 0x07;
 8                       x++;
 9               }
10               while(x < 245){
11                       *(vbuff + (y * 320) + x) = 0x0B;
```

```
12                      x++;
13              }
14              while(x < 320){
15                      *(vbuff + (y * 320) + x) = 0x07;
16                      x++;
17              }
18              y++;
19      }
20 }
```

Output



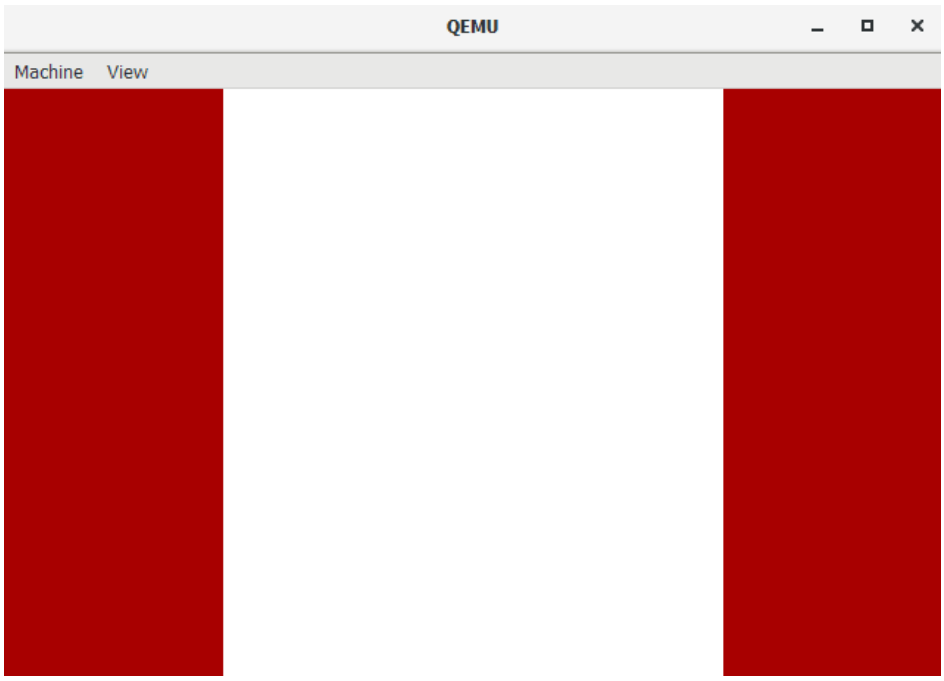
Lets see another one:

`main.c`

```
1 int start(){
2       char* vbuff = (char*) 0xA0000;
3       int y = 0;
4       while(y < 200){
5               int x = 0;
6               while(x < 75){
7                       *(vbuff + (y * 320) + x) = 0x04;
8                       x++;
9               }
10              while(x < 245){
11                      *(vbuff + (y * 320) + x) = 0x0F;
12                      x++;
```

```
13              }
14              while(x < 320){
15                      *(vbuff + (y * 320) + x) = 0x04;
16                      x++;
17              }
18              y++;
19      }
20 }
```

Output



Here , we could take the center part as the Text Editing field. This is just a sample , we wont be able to render text to the screen with any trick. We need to create our own font and make some program to render it to the screen. This may take some time , but it’s a good idea to keep working on this mode and develop an amazing UI.

Also note to work with other video modes.

External References

https://wiki.osdev.org/Drawing_In_Protected_Mode
https://wiki.osdev.org/VGA_Resources

Implementing a Mouse Driver

Introduction

`Mouse` is very essential when working with a Graphical Operating system. Although most hobbyist operating systems do not use a mouse as they work on text modes. But we need to implement it for those who are trying for a graphical OS.

For that , We will implement a driver for PS2 mouse.

How The Mouse Work

IRQ12

A `PS2` mouse generates `IRQ12`. Once we initialize a mouse , it sends 3 or 4 byte packets to communicate mouse movement, and mouse button press/release events.
We used `IRQ1` to get data from the keyboard. likewise we will be able to get mouse input by defining `IRQ12`

We could give an address of a function we create so that it will be called during every mouse input.

The Mouse Events And Packets

The first 3 bytes which the mouse give always have the same format. The second byte which it generate signifies movement in `X` axis and the third byte signifies mouse movement in `Y` axis.

When talking about the first byte , it is a combination of `8 bits` where every bit signifies different things.

The `6'th` bit of first byte is set if the middle button is clicked. `7'th` bit represents the click of Right button and `8'th` bit represents the click of Left

button.

You can use the `bit shift` and logical `and` , `or` operators in c to evaluate each of this bits.

Double Clicks

The PS2 mouse won't give you special data to represent double clicks. The best way to check if a double click occur is by looking at the time difference between first and second click. If the time difference is less(According to your need) , you could treat it as a double click.

Practical Implementation

Please obtain the full source code from:

`https://github.com/TINU-2000/OS-DEV/tree/main/Mouse`

```
unsigned char inportb(unsigned short _port){
    unsigned char rv;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
    return rv;
}

void outportb(unsigned short _port, unsigned char _data){
    __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
}

void irq_install_handler(){

     // Initialize IRQ12 in IDT and load it

}

void mouse_wait(unsigned char a_type){
     int _time_out=100000;
     if(a_type==0){
             while(_time_out--){
                     if((inportb(0x64) & 1) == 1){
                             return;
                             }
             }
             return;
     }
     else{
             while(_time_out--){
                     if((inportb(0x64) & 2) == 0){
                             return;
```

```
                                }
            }
            return;
    }
}

void ProcessMouse(){

}

void mouse_write(unsigned char a_write){
    mouse_wait(1);
    outportb(0x64, 0xD4);

    mouse_wait(1);
    outportb(0x60, a_write);
}

unsigned char mouse_read(){
    mouse_wait(0);
    return inportb(0x60);
}

void initMouse(){
    unsigned char status;
    mouse_wait(1);
    outportb(0x64, 0xA8);


    mouse_wait(1);
    outportb(0x64, 0x20);

    mouse_wait(0);
    status = (inportb(0x60) | 2);
    mouse_wait(1);
    outportb(0x64, 0x60);
    mouse_wait(1);
    outportb(0x60, status);

    // Use default settings
    mouse_write(0xF6);
    mouse_read();

    mouse_write(0xF4);
    mouse_read();

    irq_install_handler(12 , &ProcessMouse);
}

int start(){
    initMouse();
}
```

Here , the `start()` function calls a function to initialize the mouse , and in that initialize function , We communicated with the mouse to make it start

working considering the PS2 specifications. At the end of that function , we called the `irq_install_handler` function which initializes IRQ12 and load it. This function will be same as the one we used to initialize the keyboard and the only difference is the IRQ number.

After all of this is complete , whenever the mouse generates an interrupt , the cpu will start executing the `ProcessMouse()` function.

We will be able to use the `inportb` function to read from mouse and do the operation we want. So it is left to you for your own implementation!!!

External References

```
https://wiki.osdev.org/Mouse_Input
https://forum.osdev.org/viewtopic.php?t=10247
```

Audio

Introduction

Generating audio is an interesting part when doing your project. We actually give some integer value(Representing the frequency) which will turn to an audible effect.
We directly communicate with the sound card using `out` command and the sound card work on further stuff.

Let's directly start working on it!!

Generating Sound : First Try

Let's Generate a small `beep` :

`main.c`

```
unsigned char inportb(unsigned short _port){
    unsigned char rv;
    __asm__ __volatile__ ("inb %1, %0" : "=a" (rv) : "dN" (_port));
    return rv;
}

void outportb(unsigned short _port, unsigned char _data){
    __asm__ __volatile__ ("outb %1, %0" : : "dN" (_port), "a" (_data));
}

void play(unsigned int fr) {
     unsigned int Div;
     unsigned char tmp;

     Div = 1193180 / fr;
     outportb(0x43, 0xb6);
     outportb(0x42, (unsigned char) (Div) );
     outportb(0x42, (unsigned char) (Div >> 8));

     tmp = inportb(0x61);
     if (tmp != (tmp | 3)) {
             outportb(0x61, tmp | 3);
     }
}

void stop() {
```

```
27      unsigned char fr = inportb(0x61) & 0xFC;
28      outportb(0x61, fr);
29 }
30
31 void scream(int fr) {
32      play(fr);
33     int Delay = 10000;
34     while(Delay--);
35      stop();
36 }
37
38 int start(){
39      scream(2020);
40 }
```

Before compiling , add the following argument when calling qemu in `compile.bat`:

```
-soundhw pcspk
```

`compile.bat`

```
1 nasm Boot.asm -f bin -o bin\bootsect.bin
2 nasm Kernel_Entry.asm -f elf -o bin\Entry.bin
3
4 gcc -m32 -ffreestanding -c main.c -o bin\kernel.o
5
6 ld -T NUL -o bin\Kernel.img -Ttext 0x1000 bin\Entry.bin bin\kernel.o
7
8 objcopy -O binary -j .text  bin\kernel.img bin\kernel.bin
9 copy /b /Y bin\bootsect.bin+bin\kernel.bin bin\os-image
10
11 qemu-system-i386 -soundhw pcspk -drive format=raw,file=bin\os-image
```

Here , we called the `scream` function with `scream(2020)`. We could try giving different value as an argument to the scream function to generate different sound.

Generating Sound : Second Try | Integrating With OS0

Please obtain the full source code for this section from:

```
https://github.com/TINU-2000/OS-DEV/tree/main/Audio
```

Let’s include the sound generating program in our previous code. This lets us generate sound when entering a command.

This time we will feel it like a music , lets see:

extra.h

```
void setMonitorColor(char);
void cls();
void printString(char*);
void vid();
void put();
void get();

void scream(int);

extern void read();
extern void write();

unsigned char inportb(unsigned short);
void outportb(unsigned short , unsigned char);



int blockAddr;
char At[1024];

char* TM_START;

void blink(){
    setMonitorColor(0x59);
    int TIME_OUT = 0x10fffff;
    while(--TIME_OUT);
    setMonitorColor(0xa5);
}

char strcmp(char* sou , char* dest){
    int i = 0;
    while(*(sou + i) == *(dest + i)){
            if(*(sou + i) == 0 && *(dest + i) == 0)
                    return 1;
            i++;
    }
    return 0;
}

void strEval(char* CMD){
    char cmd1[] = "CLS";
    char cmd2[] = "COLORA";
    char cmd3[] = "COLORB";
    char cmd4[] = "COLORC";
    char cmd5[] = "COLORDEF";
    char cmd6[] = "VID";
    char cmd7[] = "HI";
```

```
48      char cmd8[] = "PUT";
49      char cmd9[] = "GET";
50      char cmd10[] = "PLAY";
51
52      char msg1[] = "\nHELLO , HAVE A GOOD JOURNEY LEARNING\n";
53
54      if(strcmp(CMD , cmd1))
55              cls();
56
57      else if(strcmp(CMD , cmd2))
58              setMonitorColor(0x3c);
59
60      else if(strcmp(CMD , cmd3))
61              setMonitorColor(0x5a);
62
63      else if(strcmp(CMD , cmd4))
64              setMonitorColor(0x2a);
65
66      else if(strcmp(CMD , cmd5))
67              setMonitorColor(0xa5);
68
69      else if(strcmp(CMD , cmd6))
70              vid();
71      else if(strcmp(CMD , cmd7))
72              printString(msg1);
73      else if(strcmp(CMD , cmd8)){
74              blockAddr = 0;
75              int i = 0;
76
77              while(i < 511){
78                      At[i] = 'J'; // Fill with J
79                      i++;
80              }
81              At[i] = 0; // Null character
82
83              put(); // Writes to Hard disk
84
85              i = 0;
86              while(i < 511){
87                      At[i] = 0;  // Clears the content
88                      i++;
89              }
90      }
91      else if(strcmp(CMD , cmd9)){
92              blockAddr = 0;
93              get();
94              printString(At);
95      }
96      else if(strcmp(CMD , cmd10)){
97              int stone = 15000;
98              while(stone--){
99                      scream(stone);
100             }
101     }
102
103 }
104
```

```
105 void vid(){
106     char clr = 'A';
107     while(1){
108             int i = 0;
109             while(i < 2 * 80 * 25){
110                     *(TM_START + i) = clr;
111                     clr++;
112                     i++;
113             }
114     }
115 }
116
117 void put(){
118     write();
119 }
120
121 void get(){
122     read();
123 }
124
125 void play(unsigned int fr) {
126     unsigned int Div;
127     unsigned char tmp;
128
129     Div = 1193180 / fr;
130     outportb(0x43, 0xb6);
131     outportb(0x42, (unsigned char) (Div) );
132     outportb(0x42, (unsigned char) (Div >> 8));
133
134     tmp = inportb(0x61);
135     if (tmp != (tmp | 3)) {
136             outportb(0x61, tmp | 3);
137     }
138 }
139
140 void stop() {
141     unsigned char fr = inportb(0x61) & 0xFC;
142     outportb(0x61, fr);
143 }
144
145 void scream(int fr) {
146     play(fr);
147     int stone = 0xffff;
148     while(stone--);
149     stop();
150 }
```

compile and run the program. Then type `PLAY` to play the sound.
Please note that you need to include the previous `compile.bat` file for the audio to generate. Audio will not be played if qemu is not called with `-soundhw pcspk`.

External References

https://wiki.osdev.org/PC_Speaker
https://wiki.osdev.org/Sound
https://wiki.osdev.org/Sound_Blaster_16

Going Advanced

This section is dedicated for advanced development. We will give some small descriptions for each topic and relevant reference links. Please note that this won't give you a complete idea as each topic usually requires an entire Book.

CD-ROM

ATAPI

The technology used to communicate with `CD-ROM` is `ATAPI`. It uses the Packet Interface of the `ATA6` or higher standard command set. `ATAPI` give commands needed for controlling `CD-ROM` drive and `Tape` drive.

The command set for `ATAPI` includes one for Reading and Writing Sectors , Additional features like setting `CD SPEED` , Read Track Information etc and more….

External Reference

```
https://wiki.osdev.org/ATAPI
```

USB

Universal Serial Bus

`USB` was introduced with the intention of replacing custom Hardware interfaces, and to simplify the configuration of these devices. The official `USB` specification might be hard to read and may demotivate you from implementing a driver for that. Anyway it is a `Must-To-Implement` feature if you have the plan to support USB Drives or the “Pen Drives”.

`USB` uses serial data transfer methods (And so the name `Universal Serial Bus`) and it’s different version support different transfer speeds.

External Reference

```
https://en.wikipedia.org/wiki/USB
https://wiki.osdev.org/USB
```

Networking

Networking

`Networking` also is a large topic as it involves communicating between Network interfaces(We need to implement a lot of drivers for different Interfaces including Wired and Wireless).

When a driver for these Network cards are developed , we need to implement some Program for the communication protocol like `TCP` , `UDP` etc….

External Reference

```
https://wiki.osdev.org/Intel_Ethernet_i217
https://en.wikipedia.org/wiki/Transmission_Control_Protoc
ol
https://en.wikipedia.org/wiki/User_Datagram_Protocol
```

Paging

Paging

`Paging` is a feature provided by x86 cpu which allows use of hard disk to temporarily store Data intended to be stored in ram and when needed take it back. This practically allows a program to utilize memory more than what a system have.

But this operation is little slower as a hard disk actually works at low speed. A solution for this is to use an `SSD`.

External Reference

```
https://en.wikipedia.org/wiki/Memory_paging
https://wiki.osdev.org/Paging
```

GDT

Global Descriptor Table

`Global Descriptor Table` or `GDT` is a table which contains information about memory areas and who can access those. This is very essential as an absense of this feature could allow user mode programs to do things like editing the kernel routines which results in complete system takedown.

We have already implemented gdt with the `lgdt` assembly instruction.

External Reference

```
https://en.wikipedia.org/wiki/Global_Descriptor_Table
https://wiki.osdev.org/Global_Descriptor_Table
https://wiki.osdev.org/GDT_Tutorial
```

IDT

Interrupt Descriptor Table

`Interrupt Descriptor Table` or `IDT` is a mechanism implemented in `x86` cpu which allows an operating system to capture events and do necessary operations. The events include Keyboard Input , Mouse Input etc… and also includes software defined interrupts which in x86 cpu occurs during things like when executing the `int` command.

An absense of feature like `Interrupt Descriptor Table` will make the system inefficient at working. This is because in this case , our only solution will be to ask for events every time to these devices.

External Reference

```
https://wiki.osdev.org/Interrupts
https://wiki.osdev.org/Interrupt_Descriptor_Table
https://wiki.osdev.org/Interrupt_Service_Routines
```

Timers

Programmable Interval Timer

`Programmable Interval Timer` or `PIT` is a counter that generates an interrupt when it reaches a programmed count. Things such as a `Delay Operations` could be implemented using this.

External Reference

https://en.wikipedia.org/wiki/Programmable_interval_timer
https://wiki.osdev.org/Programmable_Interval_Timer

GRUB

https://en.wikipedia.org/wiki/GNU_GRUB
https://wiki.osdev.org/GRUB

UEFI

https://www.howtogeek.com/56958/htg-explains-how-uefi-will-replace-the-bios/
https://wiki.osdev.org/UEFI
https://en.wikipedia.org/wiki/Unified_Extensible_Firmware_Interface

How To Move Further?

There are many resources online to move further. You now have an idea how things works , You can utilize it to move further. Here's some resources

```
https://littleosbook.github.io/book.pdf
http://www.brokenthorn.com/Resources/OSDevIndex.html
http://wyoos.org/Sources/index.php
```

The Thank You Summary

We have travelled a small road till here. This book aims for beginners to get started in os development and to give an idea about how things work basically. Getting good fluency in this topic need doing some projects.

Everyone have to put effort when learning a new topic. Not understanding for the first time is not a bad thing. Consistency and Patience will surely lead you to success.

At the end of time , i want every one to be successful in the Journey.
Wishing You Good Luck!!!

Thank You!!