A Report on

Development of Operating System From Ground Up

in fulfillment of the Project Elective of Operating System in

Masters of Technology (Semester 2)

in

Information Technology

Submitted By:

Tarun Jain

MT2015120

International Institute of Information Technology

IIITB , Bangalore, Karnataka

(May, 2016)

DECLARATION

I do, hereby, declare that the project elective report entitled “**Development of Operating System From Ground Up”** is an authentic work developed by me based on learning from the tutorials available at www.brokenthorn.com, under the guidance of **Prof. JayPrakash L T** and submitted for evaluation in 2nd semester as a project for the successful completion of Project Elective in Operating System at IIITB.

Tarun Jain
MT2015120
Tarun.Jain@iiitb.org

ACKNOWLEDGMENT

Before getting into the thick of the think I want to convey my best regards and few heartfelt words of gratitude towards few individuals without whom it would be a very intricate to accomplish the task of compiling a bulk of information within up to a standard content. I heartily acknowledge my project guide **Prof. JayPrakash L T** for his constant motivation and his kind notice towards the works with his valuable suggestions. I would also like to thank my mates for their valuable constant support and help me during tough times.

I duly convey a code of thanks to **Mr. Michael**, who created tutorials at **www.brokenthorn.com** which gave an enormous amount of information about an operating system and how to build an operating system from ground up. I would also like to acknowledge the efforts of students and faculties at **MIT** for creating **xv6 operating system** which helped me in many ways to get an understanding of Operating System. Last but not the least I intimate thanks to God.

Tarun Jain

ABSTRACT

Title: Development of Operating System From Ground Up

Project Description:

This project was build under the project elective course at IIITB. This project was started with an aim to learn the internal mechanism behind the complex structure of operating system. The purpose was to understand how does an OS provide such easy functionalities to user by hiding the extreme complex hardware interaction and other issues. Having said that, the project began with the basic reading of stuffs like xv6 operating system by MIT institute, and other operating system concepts. This project requires a thorough understanding of OS concepts so as to relate to the practical implementation aspects.

The project involved the study of Assembly language for 8086 Architecture. The project aims to make our own Operating System from scratch. Here, as the first version of the project is build, It only have an Boot loader which is build purely in assembly language. It will boot the operating system, loads our bootloader and display the title of our own operating system.

Tools & Technologies used:

1. Windows 7 - 32 bit
2. Bochs Emulator and Debugger
3. Virtual Floppy Disk
4. Nasm Assembler
5. Windows Debug Command
6. Notepad
7. Oracle VM Virtual Machine

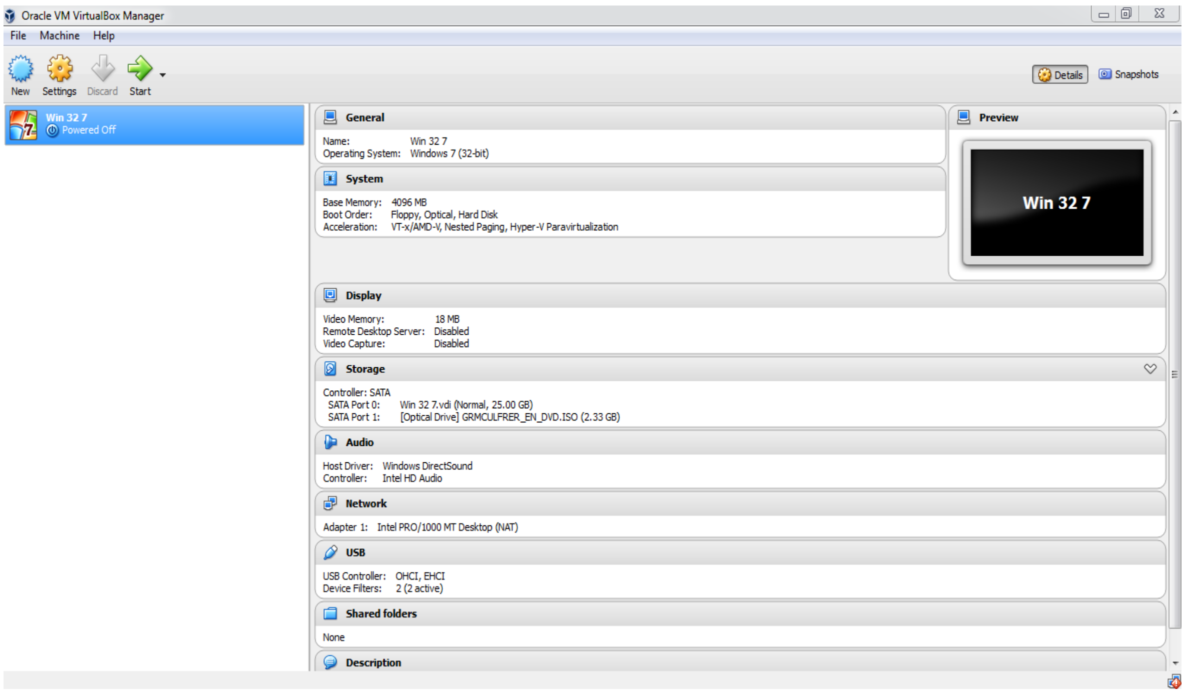
<u>**1. Introduction**</u>

As mentioned in the abstract, This report will give the details of how to make a boot loader of operating system from ground up. I have done this project by making a bootloader in four stage. Since, it is difficult to make and understand the final version of bootloader at once, we will discuss the creation of bootloader from ground up in stages. Screen shot of output and installation process is demonstrated in the steps below.

<u>**2. Setting Up System**</u>

A. Oracle VM:

First of all, Install any virtual machine on your system. I installed Oracle Virtual Machine. Then create Virtual Machine with an Image of Windows 7 - 32 bit version. It is preferred to install the 32 bit version as the other softwares which we use works best in 32 bit version operating system. Also, the windows debug command is by default available on 32 bit version without any more background installation.



General
Name: Win 32 7
Operating System: Windows 7 (32-bit)

System
Base Memory: 4096 MB
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, Hyper-V Paravirtualization

Display
Video Memory: 18 MB
Remote Desktop Server: Disabled
Video Capture: Disabled

Storage
Controller: SATA
SATA Port 0: Win 32 7.vdi (Normal, 25.00 GB)
SATA Port 1: [Optical Drive] GRMCULFRER_EN_DVD.ISO (2.33 GB)

Audio
Host Driver: Windows DirectSound
Controller: Intel HD Audio

Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT)

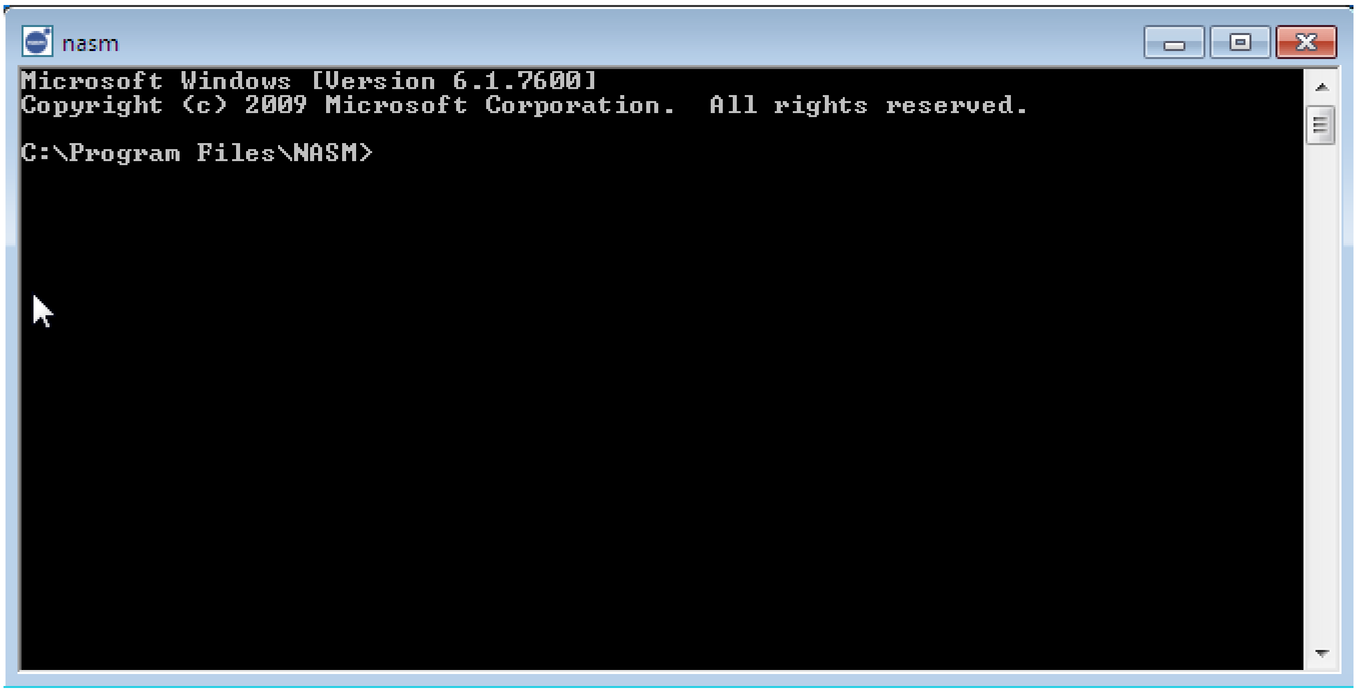
USB
USB Controller: OHCI, EHCI
Device Filters: 2 (2 active)

Shared folders
None

Description

B. NASM Assembler:

Download the NASM Assembler and install it for creating Binary Files (.bin) which is the bootloader. The NASM assembler is used to assemble our .asm files to create binary files. NASM is command line.

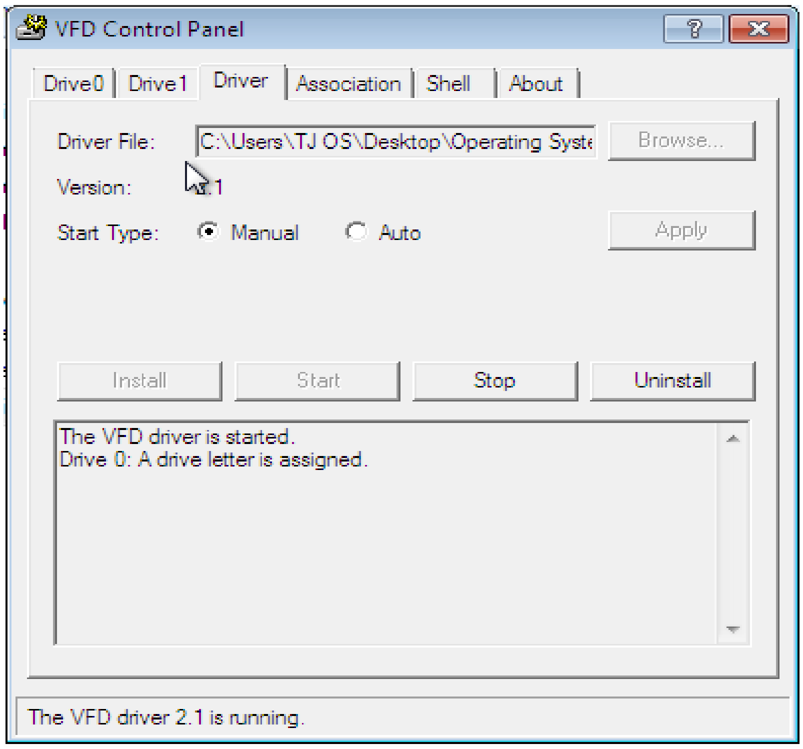


```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Program Files\NASM>
```

C. Virtual Floppy Drive:

To boot our operating system, we need to keep the bootloader in the boot sector. We do not want to disturb our system's boot sector. Thus, we will use a virtual floppy drive where we will keep our bootloader files and other necessary files. The installation process of VFD is shown below:

1. Open vfdwin.exe. RUN AS ADMINISTRATOR.
2. Under the Driver tab, Click the Start button. This starts the driver.



Driver File: C:\Users\TJ OS\Desktop\Operating Syste

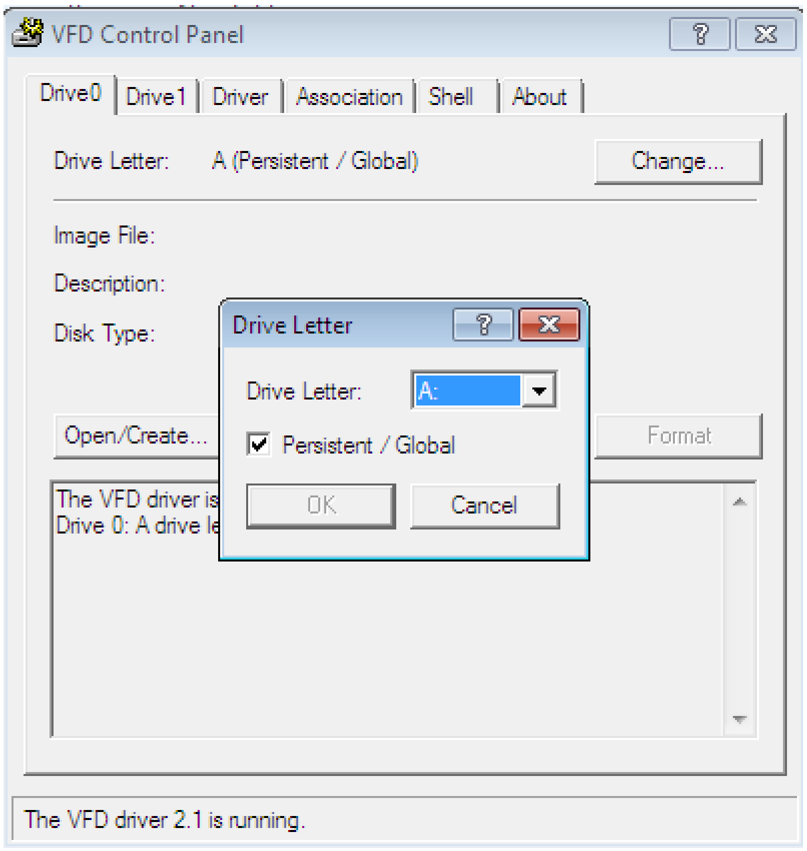
Version: 2.1

Start Type: [✓ Manual] Auto

```
The VFD driver is started.
Drive 0: A drive letter is assigned.
```

The VFD driver 2.1 is running.

3. Click either the Drive0 or Drive1 tab.



Drive Letter: A (Persistent / Global)

Image File:

Description:

Disk Type:

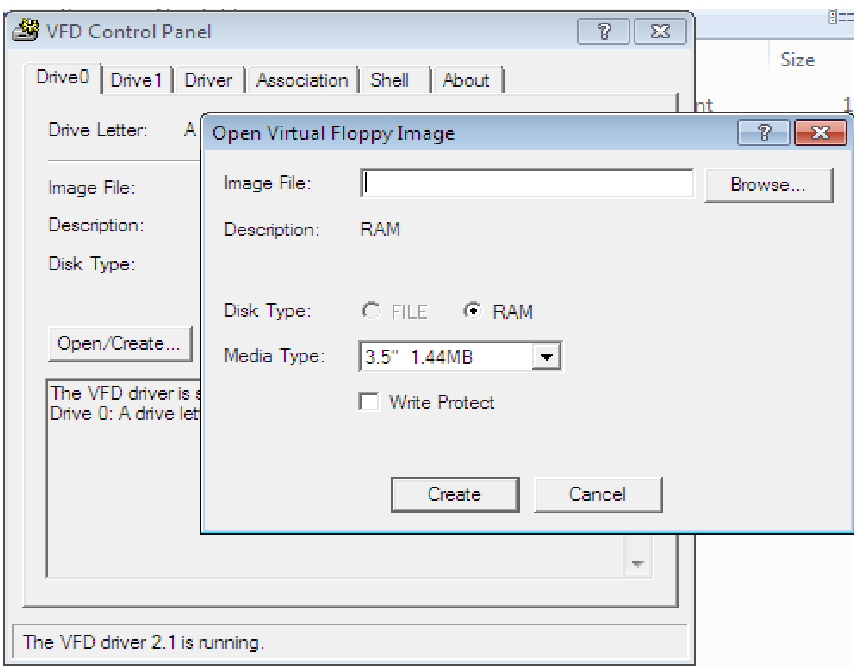
Drive Letter: A:

[✓ Persistent / Global]

```
The VFD driver is
Drive 0: A drive le
```

The VFD driver 2.1 is running.

4. Click Open/Create.


VFD Control Panel

Drive Letter: A
Image File:
Description:
Disk Type:

The VFD driver is
Drive 0: A drive let

The VFD driver 2.1 is running.

Open Virtual Floppy Image

Image File:

Description: RAM

Disk Type: FILE [✓ RAM]

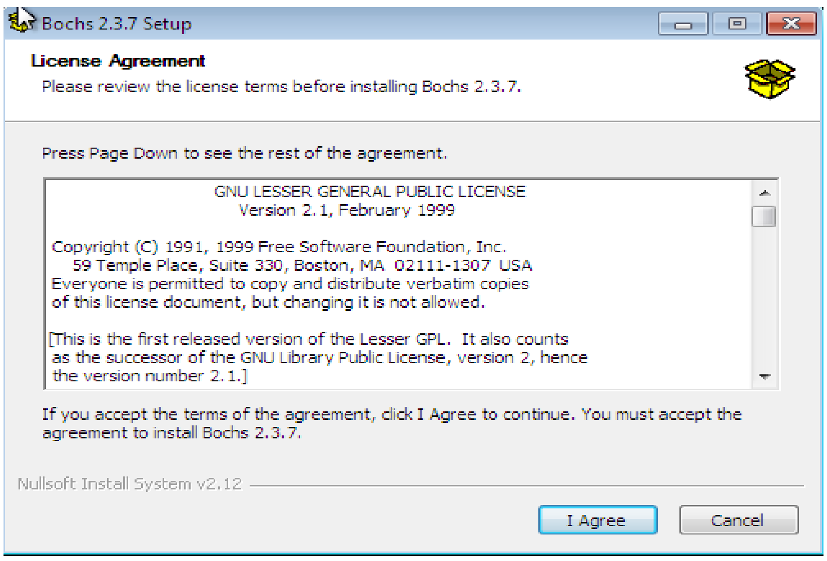
Media Type: 3.5" 1.44MB

Write Protect [ ]

Ensure Media Type is a standard 3.5" 1.44 MB floppy, and disk type is in RAM. Also, Ensure Write Protect is disabled. Click "Create".

D. Bochs Emulator and Debugger:

This is the most important software which we will use to see the output and run our own operating system. Install Bochs Emulator and leave it for now. How to use will be taught in subsequent steps.


Bochs 2.3.7 Setup

License Agreement
Please review the license terms before installing Bochs 2.3.7.

Press Page Down to see the rest of the agreement.

```
                GNU LESSER GENERAL PUBLIC LICENSE
                   Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
    59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Bochs 2.3.7.

Nullsoft Install System v2.12

E. Windows Debug Command

Go to Command Prompt in Windows. (Warning : Don't use it until instructed)

Assume: The file is at address 0x100. We want the floppy drive (Drive 0). The first sector is the first sector on the disk (sector 0) and the number of sectors is 1.

Putting this together, this is our command to write **boot_loader.bin** to the boot sector of a floppy:

```
C:\Desktop\Tarun_Jain_OS > debug boot_loader.bin
-w 100 0 0 1
-q
```

3. Development of Boot Loader

The installation of all the required softwares and getting your system ready to build our own operating system was one of the most critical phase. These are the dependencies which later create problem if not done properly in the specified manner. We will directly begin with the stage 1 of the bootloader.

A. BOOT LOADER - STAGE 1

It is assumed that, the knowledge about the hardware disk structure is already acquired by the reader. I will only explain how to build the OS, for detailed study please refer the bibliography section for references.

CLI and STI Instructions : You can use the STI and CLI instructions to enable and disable all interrupts. Most systems do not allow these instructions for applications as it can cause big problems (Although systems can emulate them).

```
cli             ; clear interrupts

; do something...

sti             ; enable interrupts--we're in the clear!
```

Double Fault Hardware Exception : If the processor finds a problem during execution (Such as an invalid instruction, division by 0, etc.) It executes a Second Fault Exception Handler (Double Fault), Which is Interrupt 0x8.If the processor still cannot continue after a double fault, it will execute a **Triple Fault**.

Triple Fault : A CPU that "Triple Faults" simply means the system hard reboots.

Developing a simple Bootloader

Bootloader ...

- Are stored with the Master Boot Record (MBR).
- Are in the first sector of the disk.
- Is the size of a single sector (512) bytes.
- Are loaded by the BIOS INT 0x19 at address 0x7C00.

Create a Notepad File: Comments explain each and every line in detail.

```
=============================================================

;*************************************************************
;       Author          : Tarun Jain
;       Roll No.: MT2015120
;       Project: Operating System Development
;       Module: Boot Loader - Version 1.0 - A Simple Bootloader
;       File     : Boot1.asm
;
;*************************************************************

org             0x7c00              ; This Bootloader is Loaded by BIOS at
                                    ; 0x7c00 address - To Tell NASM Assembler

bits            16                  ; All x86 Compatible computers boot into 16 bit
                                    ; mode
                                    ; Limits to using only 1MB of memory - Switching
                                    ; to 32 bit will be done later
Start:
                cli                 ; Clear all Interrupts
                hlt                 ; Halt the System

times           510 - ( $ - $$ ) db 0 ; Check to be within 512 bytes - Bootloader has to
                                    ; be in 1 sector
                                    ; In NASM ----$ = Address of Current Line
                                    ;             $$= Address of First Instruction
                                    ;             (here it should be 0x7c00)
                                    ; ($ - $$) = Gives the Number of Byte between
                                    ; two address
                                    ; 510 Bytes counted because Boot Signature
                                    ; instruction takes 2 Bytes, Hence total = 512
                                    ; Bytes

dw              0xAA55              ; Boot Signature
                                    ; BIOS INT 0x19 searches for a bootable disk. The
                                    ; boot signature tells that this is the bootable disk
                                    ; if 511 Byte = 0xAA and 512 Byte = 0x55 , INT
                                    ; 0x19 will load and execute the boot loader
=============================================================
```

NOTE : **In early stages, such as the bootloader, whenever there is a bug in your code, the system will triple fault. This indicates a problem in your code.**

Running Our BootLoader

Do remember this process as it will be the same process in all the stages of the bootloader. After the stage 1 - we will just learn only the changes from one stages to another in the internal working (code) of bootloader.

<1.> Save the Bootloader files as "Boot1.asm".

<2.> NASM is a command line assembler, and hence must be executed either through command line or a batch script.

To assemble **Boot1.asm** do this: (Don't forget to change directory to working directory)

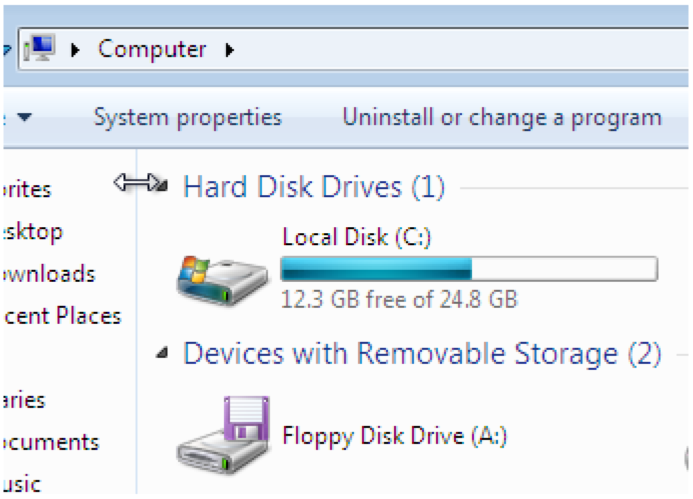
```
nasm -f bin Boot1.asm -o Boot1.bin
```

-f option : is used to tell NASM what type of output to generate. In this case, it is a binary program.

-o option : is used to give your generated file a different output name. In this case, its **Boot1.bin**

NOTE : After assembling, you should have an exact 512 byte file named "Boot1.bin".

<3.> Create a Virtual Floppy Drive as specified before. You will see a Drive named A in My Computer. Once you get it Format it. Always format it before you keep any new bootloader in drive.



<4.> Put our bootloader in the Drive using Windows Debug Command

```
C:\Desktop\Tarun_Jain_OS > debug boot1.bin
-w 100 0 0 1
-q
```

<5.> Testing the Bootloader - Using the Bochs Emulator and Debugger

** Setting Up Bochs Emulator

One time setup requires to build a configuration file for Bochs Emulator to tell about the Virtual Floppy Drive and also Specify the log file for recording errors and status of operating system.

```
#ROM and VGA BIOS images ---------------------------------------------
      romimage:    file=BIOS-bochs-latest, address=0xf0000
      vgaromimage: VGABIOS-lgpl-latest

#boot from floppy using our disk image --------------------------------
 floppya: 1_44=a:, status=inserted  # Boot from drive A

#logging and reporting -----------------------------------------------
log:      TJOS.log    # All errors and info logs will output to TJOS.log
error:    action=report
info:     action=report
```

NOTE : Save this configuration file as with extension ".bxrc" and place the configuration file in the folder where the Bochs Emulator is installed (generally C:\Program Files\Bochs2.37\)

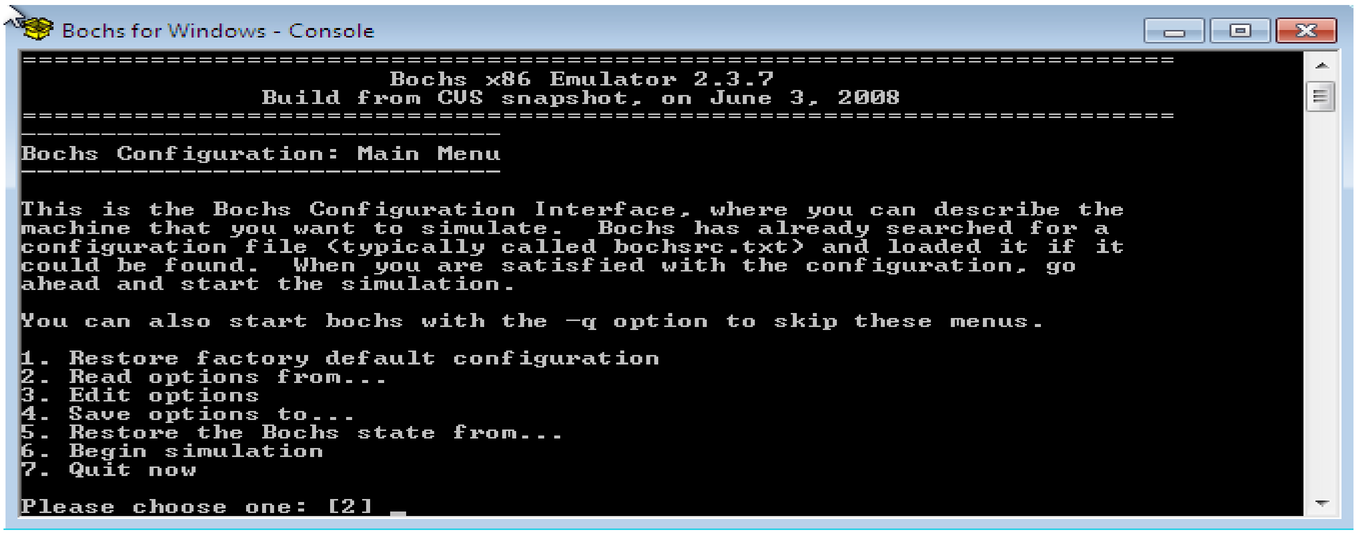
The configuration file uses # for comments. It will attempt to boot from whatever floppy disk image (Like the one we created in VFD) in drive A. The ROM BIOS and VGA BIOS images come with Bochs, so you don't need to worry about that.

If you get an error from Bochs telling you that the Bios must end at 0xFFFFF. Check the size of **BIOS-bochs-latest**, get the size in bytes (hexadecimal) and replace **address = 0xf0000** by **address=(0xfffff-size).** NOTE : Don't Forget Braces.

** How to use Bochs

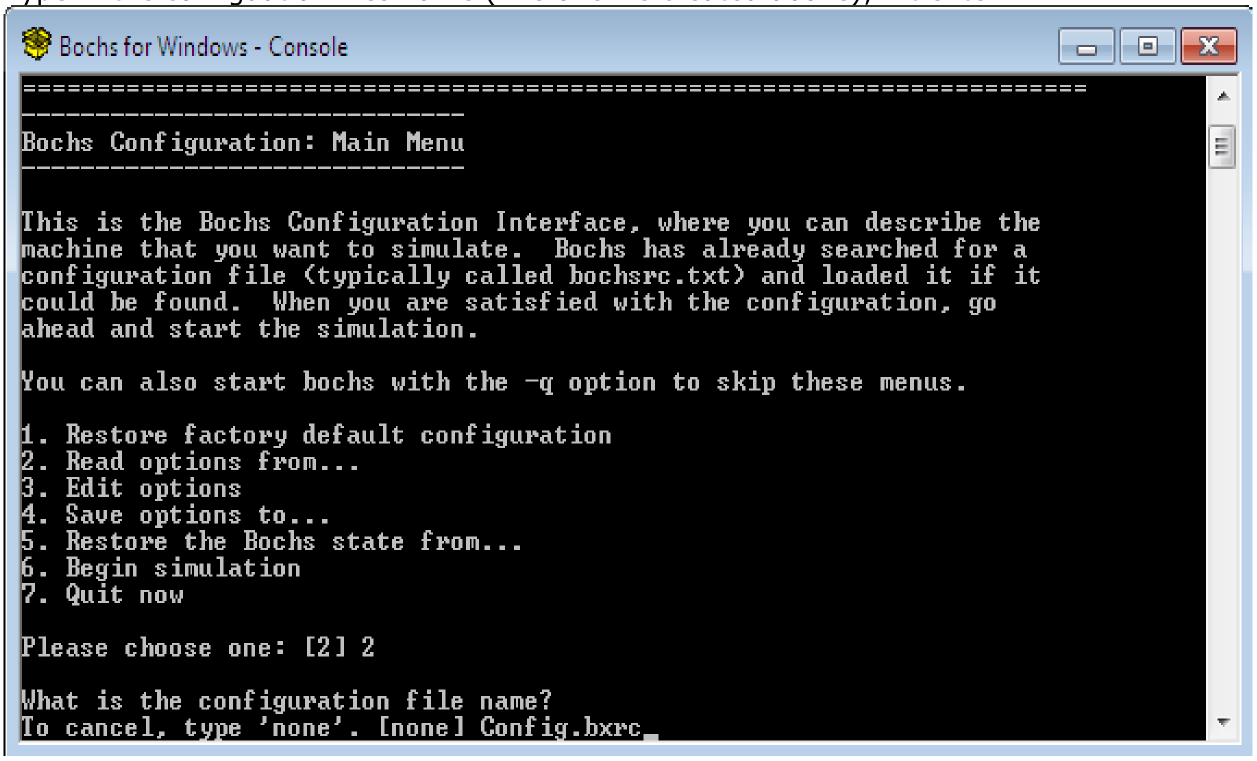
To use Bochs:

1. Execute bochs.exe
2. Select option 2 (Read options form); hit enter.



```
========================================================================
                       Bochs x86 Emulator 2.3.7
              Build from CVS snapshot, on June 3, 2008
========================================================================
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [2] _
```

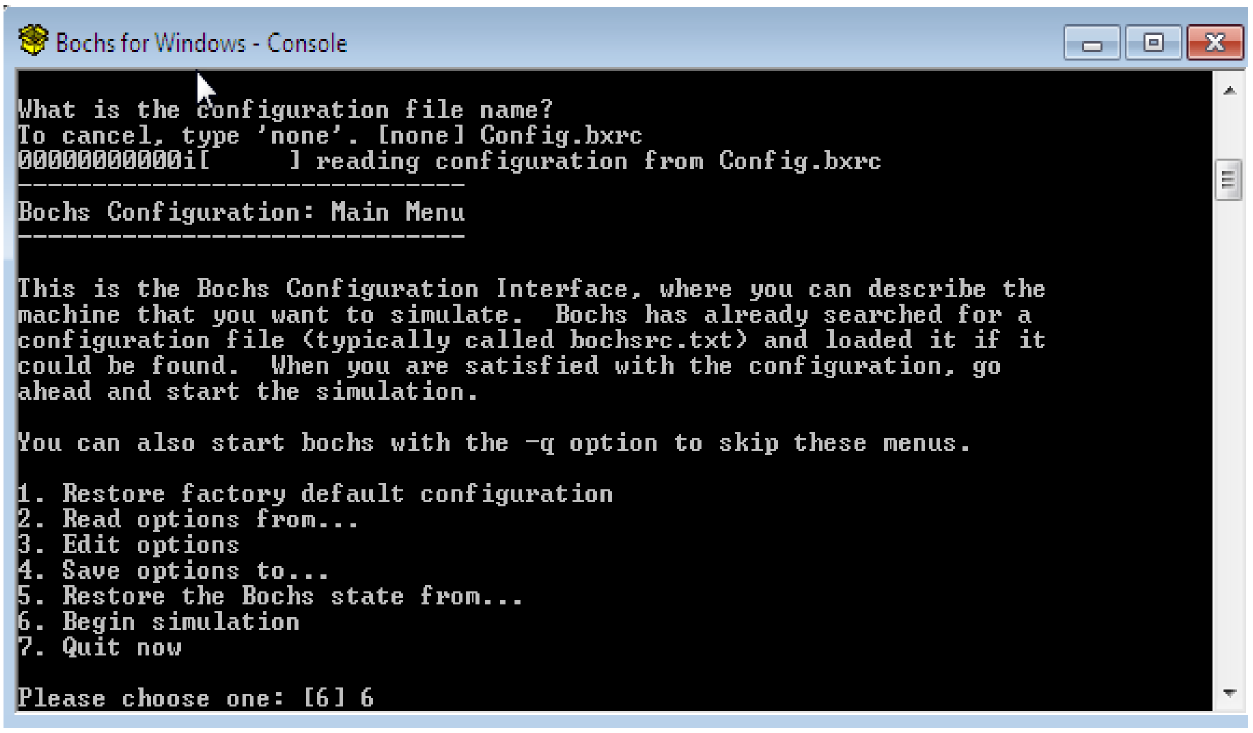
3. Type in the configuation files name (The one we created above); hit enter.



Bochs for Windows - Console

```
======================================================================
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [2] 2

What is the configuration file name?
To cancel, type 'none'. [none] Config.bxrc_
```

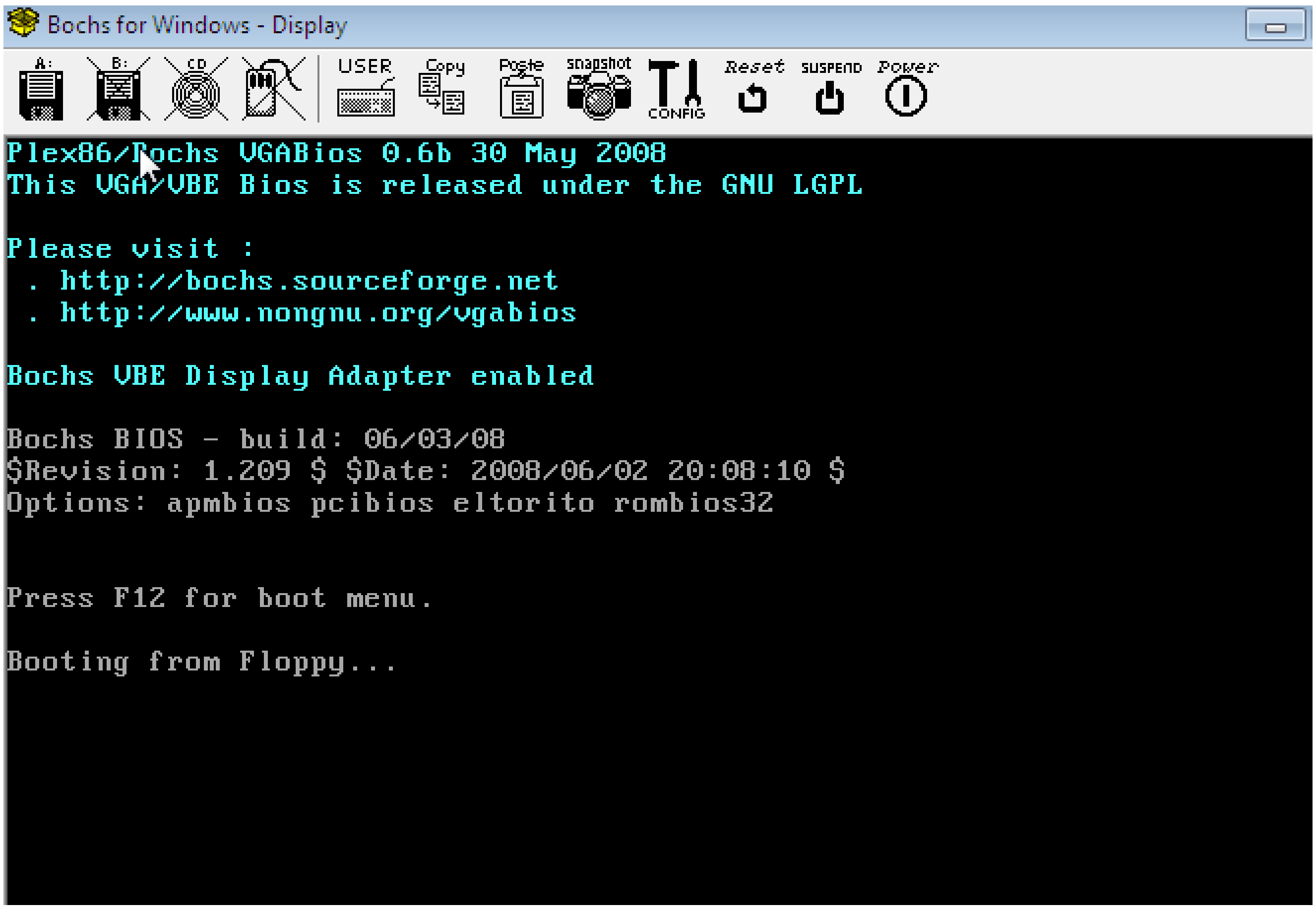
4. You will be back to the main menu. Select option 6: Begin Simulation, and hit enter.



Bochs for Windows - Console

```
What is the configuration file name?
To cancel, type 'none'. [none] Config.bxrc
00000000000i[     ] reading configuration from Config.bxrc
------------------------------
Bochs Configuration: Main Menu
------------------------------

This is the Bochs Configuration Interface, where you can describe the
machine that you want to simulate.  Bochs has already searched for a
configuration file (typically called bochsrc.txt) and loaded it if it
could be found.  When you are satisfied with the configuration, go
ahead and start the simulation.

You can also start bochs with the -q option to skip these menus.

1. Restore factory default configuration
2. Read options from...
3. Edit options
4. Save options to...
5. Restore the Bochs state from...
6. Begin simulation
7. Quit now

Please choose one: [6] 6
```

RUNNING OUTPUT OF STAGE 1 - BOOTLOADER:



```
Plex86/Bochs VGABios 0.6b 30 May 2008
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 06/03/08
$Revision: 1.209 $ $Date: 2008/06/02 20:08:10 $
Options: apmbios pcibios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...
```

==

STAGE 1 BOOTLOADER COMPLETED

===

B. BOOT LOADER - STAGE 2

Let us first see the entire code of our stage 2 bootloader and then analyze it.

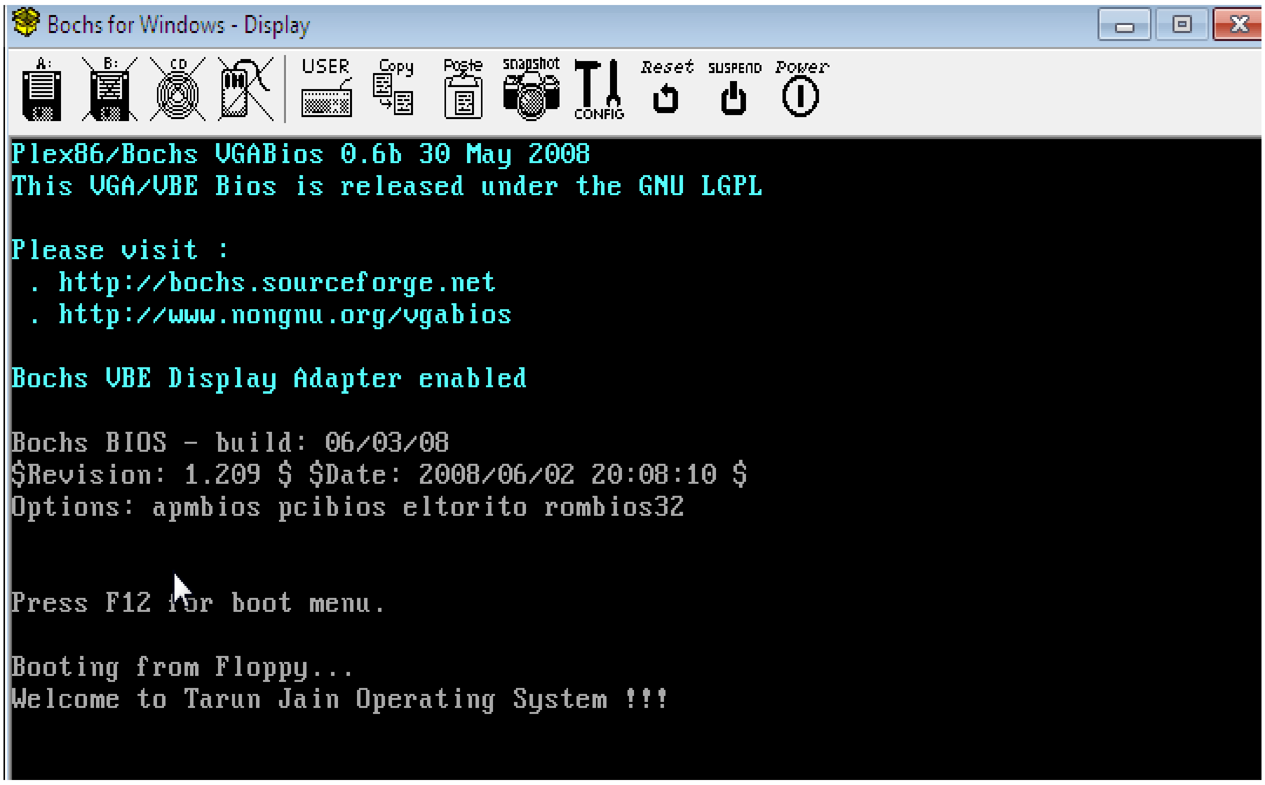
```
=============================================================
;************************************************************
;       Author          : Tarun Jain
;       Roll No.: MT2015120
;       Project: Operating System Development
;       Module: Boot Loader - Stage 2.0
;       File    : Boot1.asm
;************************************************************
bits            16                      ; We are Still in 16 bit Real Mode
org             0x7c00                  ; We are loaded by BIOS at 0x7c00
start           jmp loader              ; Jump Over OEM block
;************************************************************
;       OEM Parameter Block
;************************************************************
bpbOEM          db "Tarun OS"           ; This member must be exactly 8 bytes. It is just
                                        ; the name of your OS.
bpbBytesPerSector:          DW 512
bpbSectorsPerCluster:       DB 1
bpbReservedSectors:         DW 1
bpbNumberOfFATs:            DB 2
bpbRootEntries:             DW 224
bpbTotalSectors:            DW 2880
bpbMedia:                   DB 0xF0
bpbSectorsPerFAT:           DW 9
bpbSectorsPerTrack:         DW 18
bpbHeadsPerCylinder:        DW 2
bpbHiddenSectors:           DD 0
```

```
bpbTotalSectorsBig:      DD 0
bsDriveNumber:           DB 0
bsUnused:                DB 0
bsExtBootSignature:      DB 0x29
bsSerialNumber:          DD 0xa0a1a2a3
bsVolumeLabel:           DB "MOS FLOPPY "   ; 11 Byte
bsFileSystem:            DB "FAT12   "      ; 8 Byte

msg    db                "Welcome to Tarun Jain Operating System !!!", 0
                         ; String to print
;*************************************************************
;      Prints a string
;      DS=>SI: 0 terminated String
;*************************************************************
Print:
       lodsb             ; load next byte from string from SI to AL
       or    al,al       ; Does AL=0?
       jz    PrintDone   ; Null Terminator Found-Bail Out
       mov   ah, 0eh     ; Nope-Print the character
       int   10h         ; Call Interrupt
       jmp   Print       ; Repeat until null terminator
PrintDone:
       ret               ; We are done, so return
;*************************************************************
;      Bootloader Entry Point
;*************************************************************
loader:
       xor   ax,ax  ; Setup segments to ensure they are 0. Remember that
       mov   ds,ax  ; we have ORG 0x7c00. This means all addressess are based
       mov   es,ax  ; from 0x7c00:0. Because the data segments are within the
                    ; same code segment, null them up.
```

```
        mov     si,msg  ; our messsage to print
        call    Print   ; call out print function
        xor     ax, ax  ; clear ax
        int     0x12    ; get the amount of KB from the BIOS
        cli             ; Clear all interrupts
        hlt             ; halt the system


times 510-($-$$) db 0           ; We have to be 512 bytes. Clear the rest of the bytes
                                ; with 0

dw      0xAA55                  ; Boot Signature
```

==

RUNNING OUTPUT OF STAGE 2 - BOOTLOADER



```
Plex86/Bochs VGABios 0.6b 30 May 2008
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 06/03/08
$Revision: 1.209 $ $Date: 2008/06/02 20:08:10 $
Options: apmbios pcibios eltorito rombios32


Press F12 for boot menu.

Booting from Floppy...
Welcome to Tarun Jain Operating System !!!
```

Analysis of Code :

The OEM Parameter Block stores the Windows MBR and Boot Record information. Its primary purpose is to describe the filesystem on the disk. We will not describe this table until we look at filesystems. However, we can go no further without it. This will also fix the "Not formatted" message from Windows. For now, think of this table as a simple neccessity. I will explain it in detail later when we talk about File Systems, and loading Files off disk.

Printing Text - Interrupt 0x10 Function 0x0E

You an use INT 0x10 for video interrupts. Remember, however, that only basic interrupts will work.

INT 0x10 - VIDEO TELETYPE OUTPUT

AH = 0x0E
AL = Character to write
BH - Page Number (Should be 0)
BL = Foreground color (Graphics Modes Only)
For example:

```
        xor     bx, bx          ; A faster method of clearing BX to 0
        mov     ah, 0x0e
        mov     al, 'A'
        int     0x10
```

This will print the character 'A' on the screen.

Getting amount of RAM

INT 0x12 - BIOS GET MEMORY SIZE
Returns: AX = Kilobytes of contiguous memory starting from absolute address 0x0.

Heres an example:

```
xor     ax, ax
int     0x12
; Now AX = Amount of KB in system recorded by BIOS
```

===

STAGE 2 BOOTLOADER COMPLETED

==

C. BOOT LOADER - "STAGE 3" AND "STAGE 4"

Notice that this section contains the explanation for the both stage 3 and 4, Since stage 3 makes some changes to the bootloader but it is not a stable version which can run on its own, It can only run when the stage 4 is completed. This section will also involve some layman language for better understanding.

The Rings of Assembly Language

In Assembly Language, you might here the term "Ring 0 program", or "This program is running in Ring 3". Understanding the different rings (and what they are) can be usefull in OS Development.

Rings - Theory

Okay, so what is a ring? A "Ring", in Assembly Language, represents the level of protection and control the program has over the system. There are 4 rings: Ring 0, Ring 1, Ring 2, and Ring 3.

Ring 0 programs have absolute control over everything within a system, while ring 3 has less control. The smaller the ring number, the more control (and less level of protection), the software has.

A Ring is more then a concept--it is built into the processor arhcitecture.

When the computer boots up, even when your Bootloader executes, the processor is in Ring 0. Most applications, such as DOS applications, run in Ring 3. This means Operating Systems, as running in Ring 0, have far more control over everything then normal Ring 3 applications.

Switching Rings

Because Rings are part of the processor architecture, the processor changes states whenever it needs to. It may change when...

- A directed instruction executes a program at a different ring level, such as a far jump, far call, far return, etc.
- A trap instruction, such as INT, SYSCALL, SYSENTER
- Exceptions

We will cover Exception Handling later, as well as the SYSCALL and SYSENTER instructions.

Multi Stage Bootloaders

<u>1. Single Stage Bootloaders</u>

Remember that bootloaders, and bootsectors, are only 512 bytes in size. If the bootloader, within that same 512 bytes, executed the kernel directly, it is called a Single Stage Bootloader.

The problem with this, however, is that of its size. There is so little room to do alot within those 512 bytes. It will be very difficault to set up, load and execute a 32 bit kernel within a 16 bit bootloader. This does not include error handling code. This includes code for: GDT, IDT, A20, PMode, Loading and finding 32 bit kernel, executing kernel, and error handling. Fitting all of this code within 512 bytes is impossible. Because of this, Single stage bootloaders have to load and execute a 16 bit kernel.

Because of this problem, most bootloaders are Multi Stage Loaders.

2. Multi Stage Bootloaders

A Multi Stage Bootloader consists of a single 512 byte bootloader (The Single Stage Loader), however it just loads and executes another loader - A Second Stage Bootloader. the Second Stage Bootloader is normally 16 bit, however it will include all of the code (listed in the previous section), and more. It will be able to load and execute a 32 bit Kernel.

The reason this works is because the only 512 byte limitation is the bootloader. As long as the bootloader loads all of the sectors for the Second Stage loader in good manner, the Second Stage Loader has no limitation in size. This makes things much easier to set up for the Kernel. We will be using a 2 Stage Bootloader.

Loading Sectors Off Disk

Remember that Bootloaders are limited to 512 bytes. Because of this, there is not a whole lot we can do. As stated in the previous section, we are going to be using a 2 Stage Bootloader. This means, we will need our Bootloader to load and execute our Stage 2 program -- The Kernel Loader.

If you wanted to, The Stage 2 loader is the place to include your own "Choose your Operating System" and "Advanced Options" menu.

BIOS Interrupt (INT) 0x13 Function 0 - Reset Floppy Disk

The BIOS Interrupt 0x13 is used for disk access. You can use INT 0x13, Function 0 to reset the floppy drive. What this means is, wherever the Floppy Controller is reading from, it will immediately go to the first Sector on disk.

INT 0x13/AH=0x0 - DISK : RESET DISK SYSTEM
AH = 0x0
DL = Drive to Reset

Returns:
AH = Status Code
CF (Carry Flag) is clear if success, it is set if failure

Here is a complete example. This resets the floppy drive, and will try again if there is an error:

```
.Reset:
        mov         ah, 0           ; reset floppy disk function
        mov         dl, 0           ; drive 0 is floppy drive
        int         0x13            ; call BIOS
        jc          .Reset          ; If Carry Flag (CF) is set, there was an
                                    ; error. Try resetting again
```

Why is this interrupt important to us? Before reading any sectors, we have to insure we begin from sector 0. We dont know what sector the floppy controller is reading from. This is bad, as it can change from any time you reboot. Reseting the disk to sector 0 will insure you are reading the same sectors each time.

BIOS Interrupt (INT) 0x13 Function 0x02 - Reading Sectors

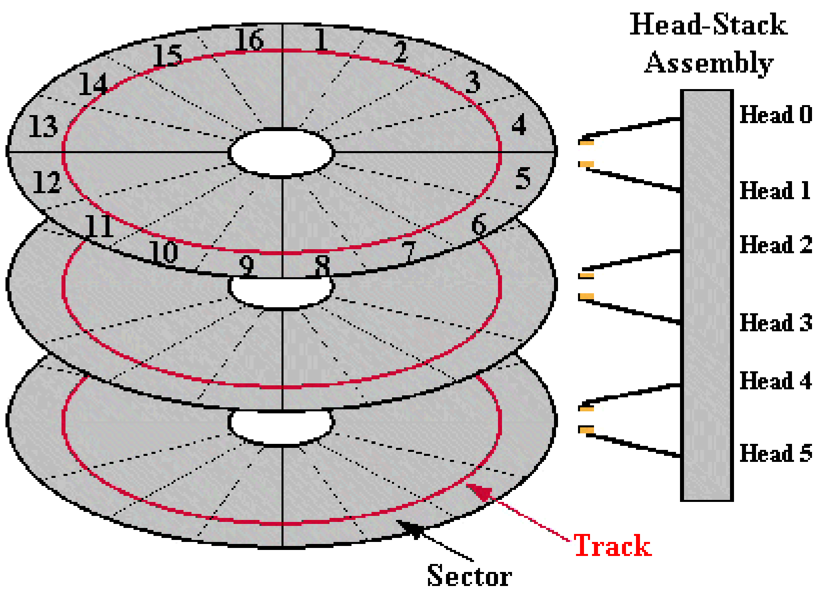
INT 0x13/AH=0x02 - DISK : READ SECTOR(S) INTO MEMORY
AH = 0x02
AL = Number of sectors to read
CH = Low eight bits of cylinder number
CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only
DH = Head Number
DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to

Returns:
AH = Status Code
AL = Number of sectors read
CF = set if failure, cleared is successfull

Okay, This is alot to think about. Some of this is fairly easy, others should be explained more. Lets take a look closer, shall we?

CH = Low eight bits of cylinder number
What is a Cylinder? A cylinder is a group of tracks (with the same radius) on the disk. To better understand this, lets look at a picture:

Drive Physical and Logical Organization



Looking at the above picture, remember:

- Each Track is useaually divided into 512 byte sectors. On floppies, there are 18 sectors per track.
- A Cylinder is a group of tracks with the same radius (The Red tracks in the picture above are one cylinder)
- Floppy Disks have two heads (Displayed in the picture)
- There is 2880 Sectors total.

What does this mean for us? The Cylinder number basically represents a track number on a single disk. In the case of a floppy disk, It represents the Track to read from.

In summary, there are 512 bytes per sector, 18 sectors per track, and 80 tracks per side.

CL = Sector Number (Bits 0-5). Bits 6-7 are for hard disks only

This is the first sector to begin reading from. Remember: There is only 18 sectors per track. This means that this value can only be between 0 and 17. You have to increase the current track number, and insure the sector number is correctly set to read the correct sector.

If this value is greater then 18, The Floppy Controller will generate an exception, because the sector does not exist. Because there is no handler for it, The CPU will generate a second fault exception, which will ultimately lead to a Triple Fault.

DH = Head Number

Remember that some floppys have two heads, or sides, to them. Head 0 is on the front side, where sector 0 is. Because of this, We are going to be reading from Head 0.

If this value is greater then 2, The Floppy Controller will generate an exception, because the head does not exist. Because there is no handler for it, The CPU will generate a second fault exception, which will ultimately lead to a Triple Fault.

DL = Drive Number (Bit 7 set for hard disks)
ES:BX = Buffer to read sectors to

What is a Drive Number? Its simply a number that, of course, represents a drive. Drive Number 0 useually represents a floppy drive. Drive Number 1 is useually for 5-1/4" Floppy drives.

Because we are on a floppy, we want to read from the floppy disk. So, the drive number to read from is 0.

ES:BX stores the segment:offset base address to read the sectors into. Remember that the Base Address represents the starting address.

Reading and loading a sector

To read a sector from disk, first reset the floppy drive, and just read:

```
.Reset:
        mov         ah, 0           ; reset floppy disk function
        mov         dl, 0           ; drive 0 is floppy drive
        int         0x13            ; call BIOS
        jc          .Reset          ; If Carry Flag (CF) is set, there's error
        mov         ax, 0x1000      ; to read sector to address 0x1000:0
        mov         es, ax
        xor         bx, bx

.Read:
        mov         ah, 0x02        ; function 2
        mov         al, 1           ; read 1 sector
        mov         ch, 1           ; we still on track 1
        mov         cl, 2           ; sector to read (The second sector)
        mov         dh, 0           ; head number
        mov         dl, 0           ; drive number.
```

```
          int                 0x13                ; call BIOS - Read the sector
          jc                  .Read               ; Error, so try again
          jmp                 0x1000:0x0          ; jump to execute the sector!
```

Note: If there is a problem reading the sectors, and you try to jump to it to execute it, The CPU will exeute whatever instructions at that address, weather or not the sector was loaded. This useually means the CPU will run into either an invalid/unkown instruction, or the end of memory, both will result in a Triple Fault.

The above code only reads and executes a raw sector, which is kind of pointless to our needs. For one,We currently have PartCopy set up to copy only 512 bytes, which means: Where and how are we going to create a raw sector?

Also, it is impossible for us to give this Raw Sector a "filename" because it does not exist. Its just a raw sector.

Finally, We currently have the bootloader setup for a FAT12 File System. Windows will attempt to read certain tables (File Allocation Tables) from Sector 2 and 3. However, with a Raw Sector, these tables are nonexistant, so Windows will take garbage values (as if it was the table). The result? When reading the floppy disk from Windows, you will see files and directories with currupt names, and enormous sizes (Have you ever seen a 2.5 Gigabyte file on a 3.14 MB Floppy? I have :))

Of course, We Will need to load sectors this way. Before we do, however, we have to find the Starting Sector, Number of sectors, base address, etc. of a file in order to load it properly. This is the bases of loading files off disk.

Navigating The FAT12 FileSystem

OEM Parameter Block - Detail

In the previous artical, we dumped an table in our code. What was it? Oh yeah...

```
bpbBytesPerSector:          DW 512
bpbSectorsPerCluster:       DB 1
bpbReservedSectors:         DW 1
bpbNumberOfFATs:            DB 2
bpbRootEntries:             DW 224
bpbTotalSectors:            DW 2880
bpbMedia:                   DB 0xF0
bpbSectorsPerFAT:           DW 9
bpbSectorsPerTrack:         DW 18
bpbHeadsPerCylinder:        DW 2
bpbHiddenSectors:           DD 0
bpbTotalSectorsBig:         DD 0
bsDriveNumber:              DB 0
bsUnused:                   DB 0
bsExtBootSignature:         DB 0x29
bsSerialNumber:             DD 0xa0a1a2a3
bsVolumeLabel:              DB "MOS FLOPPY "
bsFileSystem:               DB "FAT12   "
```

Alot of this is pretty simple. Lets analyze this in some detail:

```
bpbBytesPerSector:          DW 512
bpbSectorsPerCluster:       DB 1
```

bpbBytesPerSector indicates the number of bytes that represents a sector. This must be a power of 2. Normally for floppy disks, it is 512 bytes.

bpbSectorsPerCluster indicates the number of sectors per cluster. In our case, we want one sectorper cluster.

```
bpbReservedSectors:         DW 1
bpbNumberOfFATs:                DB 2
```

A Reserved Sector is the number of sectors not included in FAT12. ie, not part of the Root Directory. In our case, The Bootsector, which contains our bootloader, will not be part of this directory. Because of this, bpbReservedSectors should be 1.

This also means that the reserved sectors (Our bootloader) will not contain a File Allocation Table.

bpbNumberOfFATs rpresents the number of File Allocation Tables (FATs) on the disk. The FAT12 File System always has 2 FATs.

Normally, you would need to create these FAT tables. However, Because we are using VFD, We can have Windows/VFD to create these tables for us when it formats the disk.

Note: These tables will also be written to by Windows/VFD when you add or delete entrys. ie, when you add a new file or directory.

```
bpbRootEntries:       DW 224
bpbTotalSectors:      DW 2880
```

Floppy Disks have a maximum of 224 directories within its Root Directory. Also, Remember that there are 2,880 sectors in a floppy disk.

```
bpbMedia:                      DB 0xF0
bpbSectorsPerFAT:                   DW 9
```

The Media Descriptor Byte (bpbMedia) is a byte code that contains information about the disk. This byte is a Bit Pattern:

- Bits 0: Sides/Heads = 0 if it is single sided, 1 if its double sided
- Bits 1: Size = 0 if it has 9 sectors per FAT, 1 if it has 8.
- Bits 2: Density = 0 if it has 80 tracks, 1 if it is 40 tracks.
- Bits 3: Type = 0 if its a fixed disk (Such as hard drive), 1 if removable (Such as floppy drive)
- Bits 4 to 7 are unused, and always 1.

0xF0 = 11110000 binary. This means it is a single sided, 9 sectors per FAT, 80 tracks, and is a movable disk. Look at bpbSectorsPerFAT and you will see that it is also 9.

```
bpbSectorsPerTrack:         DW 18
bpbHeadsPerCylinder:        DW 2
```

Remember: from the previous tutorials/ There is 18 sectors per track. bpbHeadsPerCylinder simply represents that there are 2 heads that represents a cylinder. (If you dont know what a Cylinder is, please read the section "BIOS Interrupt (INT) 0x13" on Reading Sectors.)

```
bpbHiddenSectors:             DD 0
```

This represents the number of sectors from the start of the physical disk and the start of the volume.

```
bpbTotalSectorsBig:    DD 0
bsDriveNumber:         DB 0
```

Remember that the floppy drive is Drive 0?

```
bsUnused:                 DB 0
bsExtBootSignature:        DB 0x29
```

The Boot Signiture represents the type and version of this BIOS Parameter Block (This OEM Table) is. The values are:

- 0x28 and 0x29 indicate this is a MS/PC-DOS version 4.0 Bios Parameter Block (BPB)

We have 0x29, so this is the version we are using.

```
bsSerialNumber:        DD 0xa0a1a2a3
bsVolumeLabel:         DB "MOS FLOPPY "
bsFileSystem:          DB "FAT12   "
```

The serial number is assigned by the utility that formats it. The serial number is unique to that particular floppy disk, and no two serial numbers should be identical.

Microsoft, PC, and DR-DOS base the Seral number off of the current time and date like this:

```
Low 16 bits = ((seconds + month) << 8) + (hundredths + day_of_month)

High 16 bits = (hours << 8) + minutes + year
```

Because the serial number is overwritten, we could put whatever we want in it--it doesnt matter.

The Volume Lable is a string to indicate what is on the disk. Some OSs display this as its name. Note: This string *Must* be 11 bytes. No more, and no less.

The Filesystem string is used for the same purpose, and no more. Note: This string *must* be 8 bytes, no more and no less.

FAT12 Filesystem - Theory

FAT12 is the first FAT (File Allocation Table) Filesystem released in 1977, and used in Microsoft Disk BASIC. FAT12, as being an older filesystem generally released for floppr disks, had a number of limtations.

- FAT12 has no support for hierarchical directories. This means there is only one directory-**Thr Root Directory**.

- Cluster Addresses were only 12 bits long, which limits the maximum number of clusters to 4096

- **The Filenames are stored in the FAT as a 12 bit identifier. The Cluster Addresses represent the starting clusters of the files.**

- Because of the limited cluster size, **The maximum number of files possible is 4,077**

- The Disk Size is stored only as a 16 bit count of sctors, limiting it to 32 MiB in size
- FAT12 uses the value "0x01" to identify partitions

These are some big limitations. Why do we want FAT12 then?

FAT16 has support for directories, and over 64,000 files as it uses a 16 bit cluster (file) address, as apposed to FAT16. However, **FAT16 and FAT12 are very simular.**

To make things simple, we are going to use FAT12. We might spruce things up with FAT16 (or even use FAT32) later :) (FAT32 is quite different then FAT 12/16, so we might ust use FAT16 later.)

FAT12 FileSystem - Disk Storage

To understand more about FAT12, and how it works, it is better to look at the structure of a typical formatted disk.

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.

This is a typical formatted FAT12 disk, from the bootsector to the very last sector on disk.

Step 1: Loading the Root Directory Table

Now its time to load Stage2.sys! We will be refrencing the Root directory table alot here, along with the BIOS parameter block for disk information.

Step 1: Get size of root directory

Okay, first we need to get the size of the root directory.

To get the size, just multiply the number of entrys in the root directory. Seems simple enough :)

In Windows, whenever you add a file or directory to a FAT12 formatted disk, Windows automatically adds the file information to the root directory, so we dont need to worry about it. This makes things much simpler.

Dividing the number of root entrys by bytes per sector will tell us how many sectors the root entry uses.

Here is an example:

```
          mov     ax, 0x0020                 ; 32 byte directory entry
          mul     WORD [bpbRootEntries]      ; number of root entrys
          div     WORD [bpbBytesPerSector]   ; get sector used by root dir
```

Remember that the root directory table is a table of **32 byte values (entrys)** that reprsent the file information.

Yippe--Okay, we know how much sectors to load in for the root directory. Now, lets find the starting sector to load from :)

Step 2: Get start of root directory

This is another easy one. First, lets look at a FAT12 formatted disk again:

Boot Sector	Extra Reserved Sectors	File Allocation Table 1	File Allocation Table 2	Root Directory (FAT12/FAT16 Only)	Data Region containng files and directories.

Okay, note that the **Root Directory is located directly after both FATs and reserved sectors**. In other words, just add the FATs + reserved sectors, and you found the root direcory!

For example...

```
mov      al, [bpbNumberOfFATs]  ; Get number of FATs (Useually 2)
mul      [bpbSectorsPerFAT]     ; number of FATs * sectors per FAT
                                ; get number of sectors
add      ax, [bpbReservedSectors] ; add reserved sectors

                       ; Now, AX = starting sector of root directory
```

Now, we just read the sector to some location in memory:

```
          mov      bx, 0x0200   ; load root directory to 7c00:0x0200
          call     ReadSectors
```

Step 2: Find Stage 2

Okay, now the root directory table is loaded. Looking at the above code, **we loaded it to 0x200.** Now, to find our file.

Lets look back at the 32 byte root directory table again (Section **Root Directory Table**. Remember **the first 11 bytes represent the filename**. Also remember that, **because each root directory entry is 32 bytes, Every 32 bytes will be the start of the next entry - Pointing us back to the first 11 bytes of the next entry.**

Hence, all we need to do is compare filenames, and jump to the next entry (32bytes), and test again until we reach the end of the sector. For example...

```

     ; browse root directory for binary image
mov      cx, [bpbRootEntries]    ; the number of entrys
mov      di, 0x0200             ; Root directory was loaded here

.LOOP:
          push     cx
          mov      cx, 11            ; eleven character name
          mov      si, ImageName     ; compare the 11 bytes with the name
of our file
          push     di
          rep   cmpsb                     ; test for entry match
          pop      di
          je       LOAD_FAT          ; they match, so begin loading FAT
          pop      cx
          add      di, 32            ; they dont match, so go to next
entry (32 bytes)
          loop     .LOOP
          jmp      FAILURE           ; no more entrys left, file doesnt
exist :(
```

Step 3: Loading FAT

Step 1: Get start cluster

Okay, so the root directory is loaded and we found the files entry. How do we get its starting cluster?

- Bytes 26-27 : First Cluster
- Bytes 28-32 : File Size

This should look familiar :) To get the starting cluster, just refrence byte 26 in the file entry:

```
mov     dx, [di + 0x001A]   ; di contains starting address of entry.
Just refrence byte 26 (0x1A) of entry

; Yippe--dx now stores the starting cluster number
```

The starting cluster will be important to us when loading the file.

Step 2: Get size of FAT

Lets look at the BIOS parameter block again. More specifically...

```
bpbNumberOfFATs:        DB 2
bpbSectorsPerFAT:       DW 9
```

Okay, so how do we find out how many sectors there are in both FATs? Just multiply sectors per FAT by the number of sectors.

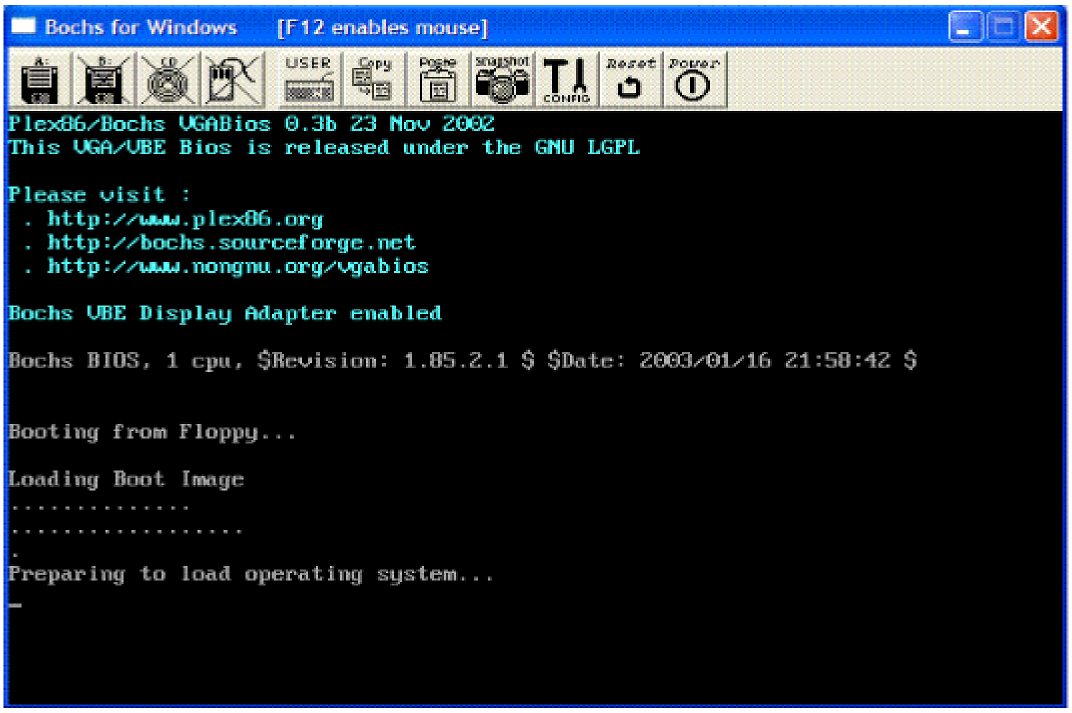
```
          xor     ax, ax
          mov     al, [bpbNumberOfFATs]              ; number of FATs
          mul     WORD [bpbSectorsPerFAT]            ; multiply by
number of sectors per FAT

        ; ax = number of sectors the FATs use!
```

Step 3: Load the FAT

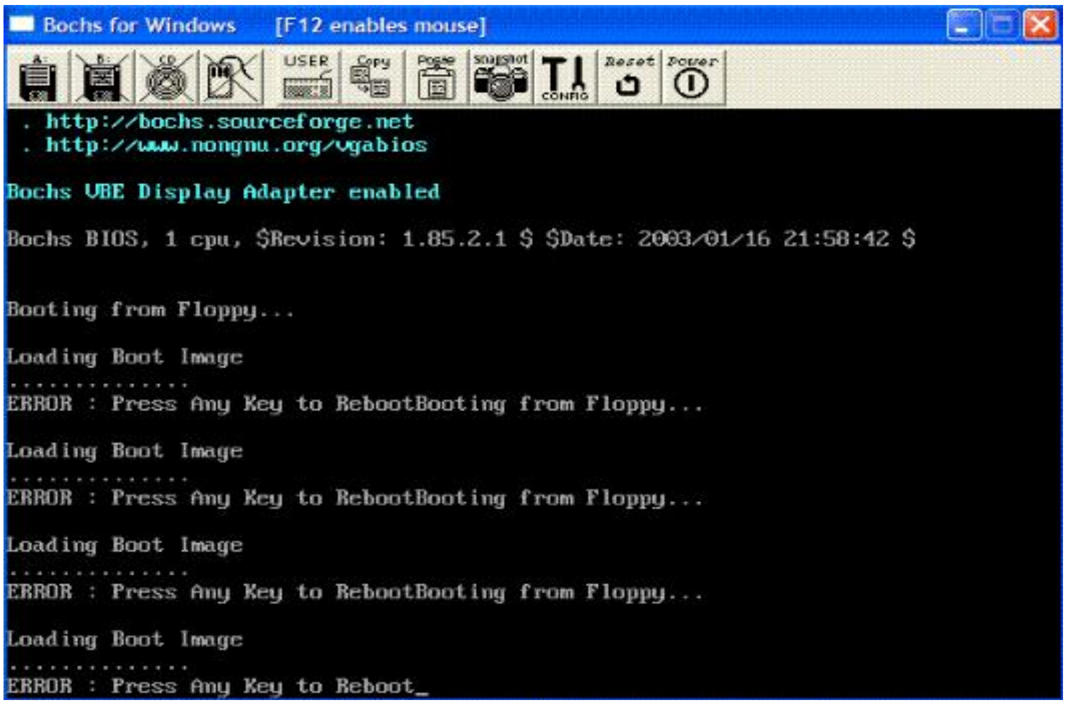
Now that we know how many sectors to read.

```
          mov     bx, 0x0200                        ; address to load
to
          call    ReadSectors                       ; load the FAT
table
```

RUNNING OUTPUT OF STAGE 3 AND STAGE 4 BOOT LOADER:



```
Plex86/Bochs VGABios 0.3b 23 Nov 2002
This VGA/VBE Bios is released under the GNU LGPL

Please visit :
 . http://www.plex86.org
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS, 1 cpu, $Revision: 1.85.2.1 $ $Date: 2003/01/16 21:58:42 $


Booting from Floppy...

Loading Boot Image
..............
.................
.
Preparing to load operating system...
_
```



```
 . http://bochs.sourceforge.net
 . http://www.nongnu.org/vgabios

Bochs VBE Display Adapter enabled

Bochs BIOS, 1 cpu, $Revision: 1.85.2.1 $ $Date: 2003/01/16 21:58:42 $


Booting from Floppy...

Loading Boot Image
..............
ERROR : Press Any Key to RebootBooting from Floppy...

Loading Boot Image
..............
ERROR : Press Any Key to RebootBooting from Floppy...

Loading Boot Image
..............
ERROR : Press Any Key to RebootBooting from Floppy...

Loading Boot Image
..............
ERROR : Press Any Key to Reboot_
```

===

STAGE 3 and 4 BOOTLOADER - PROJECT COMPLETED

==

4. Future Enhancement & Conclusion

This project builds a full working bootloader which loads a entire second stage bootloader which in turn can access the other sectors of the disk. This project can now be extended to load a kernel from the disk. For this purpose we need to make kernel and system calls associated with it. The extension of this project has undefined scope as an operating system can be added with many functionality to make it more mature.

5. Bibliography

[1.] http://www.brokenthorn.com
[2.] Assembly Language - Step by Step
[3.] Art of Assembly
[4.] MIT xv6 Operating System
[5.] www.youtube.com
[6.] www.google.co.in