**PROJECT REPORT**

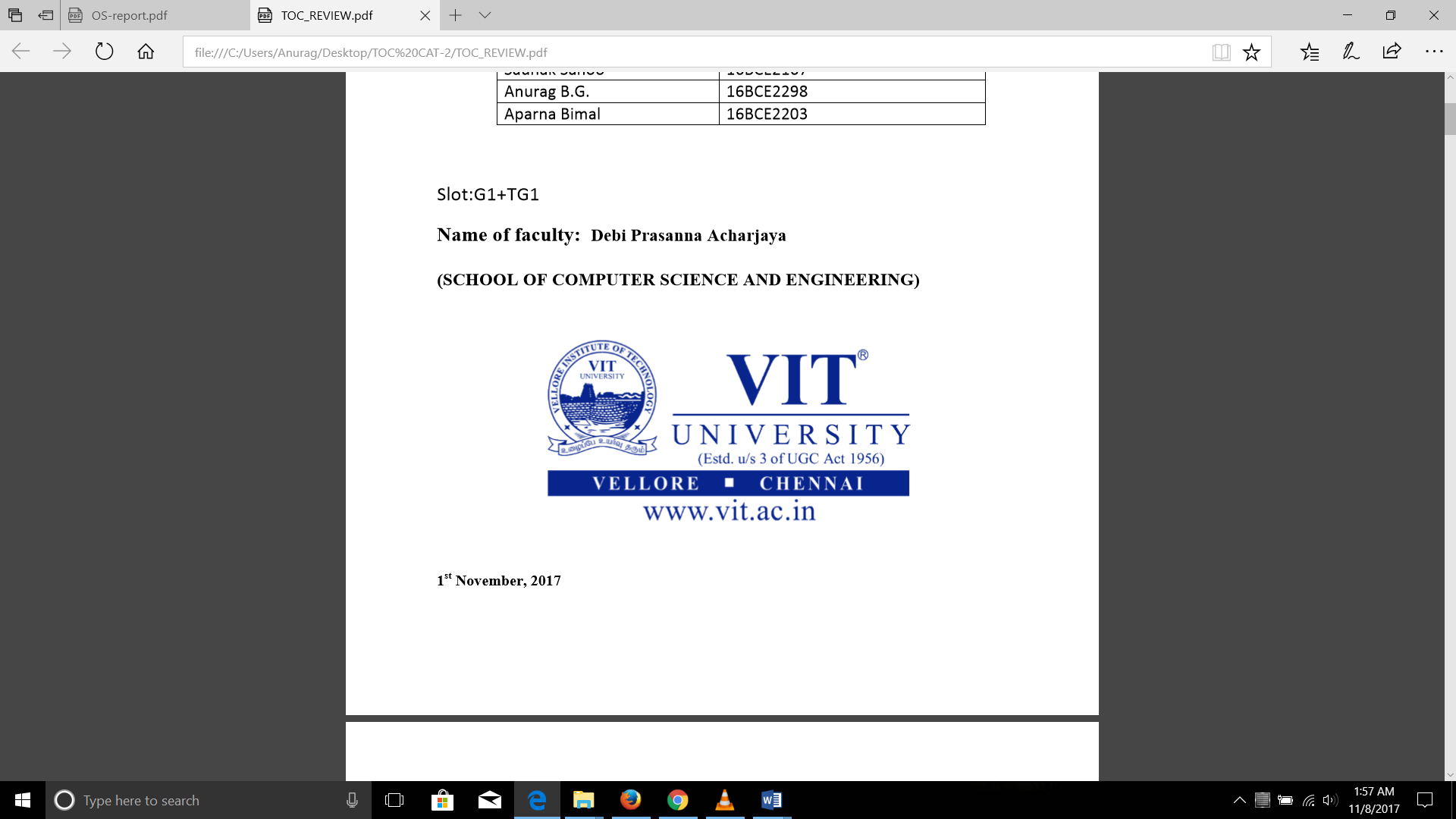
***OPERATING SYSTEM***

***A SIMPLE BOOTLOADER EXECUTION***

**FINAL REVIEW REPORT**

Submitted by:-

|  |  |
| --- | --- |
| **NAME** | **REGISTRATION NUMBER** |
| *Shaik Muzammil* | *16BCE0481* |
| *Anurag B.G.* | *16BCE2298* |
| *Corrina Marieanne Barnabas* | *16BCE0754* |
| *Saunak Sahoo* | *16BCE2167* |



**CERTIFICATE**

This is to certify that the project work entitled “A simple bootloader execution” that is being Submitted for

OPERATING SYSTEMS (CSE2005) is a record of bonafide work done under my supervision. The contents of this project work, in full or in parts, have neither been taken from any other Source nor have been submitted for any other CAL course.

Place: VIT University, Vellore

Date: 8th November, 2017

Signature of Faculty: **SHAIK NASEERA**

**ACKNOWLEDGEMENTS**

We thank our Dean, for the well-crafted education pattern due to which undertaking of this project was possible. Next, we thank our **Prof, Shaik Naseera** for always encouraging us to perform better and giving us the knowledge of the subject that played an important role in completion of project. We also thank our classmates and friends who helped us debug some of the problems. We are grateful to our parents too for sending us to such a good Institution like VIT

**ABSTRACT**

**Bootloader** is a piece of code/program that runs before an operating system starts to run. It loads an operating system when a computer is turned on. It tells the hardware where to look and how to get running when you start things up.

**Bootloader** basically package the instructions to boot operating system kernel and most of them also have their own debugging or modification environment. As the **bootloader** kicks off before any piece of software on your device, it makes it extremely processor specific and every motherboard has its own **bootloader**. This is the reason all android devices have different custom ROM’s due to their varying hardware specifications.

The **bootloader** is the program that is responsible for checking and loading the operating system. Hence manufacturers roll out their devices with locked bootloader so that some users who don’t know what they’re doing don’t damage their devices with bad software. Hence, we need to unlock the bootloader first in Android devices before installing any custom ROM.

The **bootloader** configures the device to an initial known state and has a means to select where to start executing the kernel. It can allow you to make this selection, which allows the user, among other things, to load an alternative Linux kernel, or Windows. Because the **bootloader** is an essential component of the boot process, it is stored in non-volatile memory, such as flash memory.

**Bootloaders** are written by hardware vendors and are specialized for the hardware they run on.

To write a **bootloader**, First, we should understand the memory map of the Flash and decide where the **bootloader** sits and where the application sits (some controllers have dedicated boot section) and the finalize the interface and then specify a protocol to transfer the application code via the interface and get it to the **bootloader** and then write it to the flash.

**INTRODUCTION**

A **Bootloader** is the first program which executes (before the main program) whenever a system is initialized. In PC It is used to boot he OS of the system. In case of *microcontrollers*, **a bootloader** enriches the capabilities of the microcontroller and makes them self-programmable device. This article will explore the significance of having **bootloader in microcontroller**.

The ***bootloader***can be inserted into a controller by using an external or any conventional burner and then depending on the type of bootloader the controller starts responding to the interface. So, whenever the controller is reinitialized the program counter jumps to the bootloader section and then it waits there for the instruction, which is fed from external device. Boot loader is only run when the boot loader is installed on a disk

The **bootloader** is the code that loads the operating system. When a CPU comes out of reset (ie. power-on) it always starts executing code at a certain memory address. That code must be contained in non-volatile memory, and may do any number of things before eventually loading into memory an OS, configuring some kind of filesystem for the OS to use, and finally launching the OS kernel. Once the kernel has been invoked, the role of the bootloader is over until the system reboots.

We may or may not be able to craft a bootloader for a device we already own, depending on the ability of the device to be reprogrammed, or to use a progressive multi-stage boot process. In some systems, the non-volatile code that is invoked at reset is permanent and immutable from manufacture. Sometimes, it can be replaced using special tools such as a J-tag programmer. Sometimes, there is a process that provides for third party extensions to the bootloader to be written. The PC BIOS is such a system, where the BIOS loads a bootloader from either a known standard location on a bootable media (spinning disk, flash disk, etc), or from a network using PXE. This allows different bootloaders to be written and installed, such as the Windows bootloader, grub, lilo, syslinux, etc.

**Bootloaders** are programs which can program a program. They sit within the flash and gets the "application code" of user via some means (say SPI or I2C or Over the air) and put it into some location in the flash (different from where it is sitting) and then finally can do some sanity check on the placed application code to know whether it is valid and then let the code execute.

**Methodology**

In this project we have built 2 modules to demonstrate the working of a bootloader.

Both modules have certain common files:

**Test.c** : Assembly language written with the help of C language which contains the bootloader program

The above code is executed using an **assembler** to generate machine language (object code).

**Test.o:** Object code for bootloader program

**Test.ld**: Linker file

The .ld file is linked to the object file using a **linker** to generate an executable program

**Floppy (disc image file):** The executable file which is loaded by the **BIOS** software from memory and executed at bootup

**BOCHS:** A software used to simulate the action of the BIOS to load 16-bit bootloader program and display output

**MODULE-1:** Printing ‘Hello World’

Test.c file:

/\*generate 16-bit code\*/

\_\_asm\_\_(".code16gcc\n");

/\*jump boot code entry\*/

\_\_asm\_\_("jmpl $0x0000, $\_\_main\n");

void \_main() {

/\*print letter 'H' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'H' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'e' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'e' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'l' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'l' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'l' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'l' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'o' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'o' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter ',' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $',' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter ' ' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $' ' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'W' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'W' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'o' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'o' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'r' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'r' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'l' onto the screen\*/

\_\_asm\_\_ \_\_volatile\_\_("movb $'l' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

/\*print letter 'd' onto the screen\*/

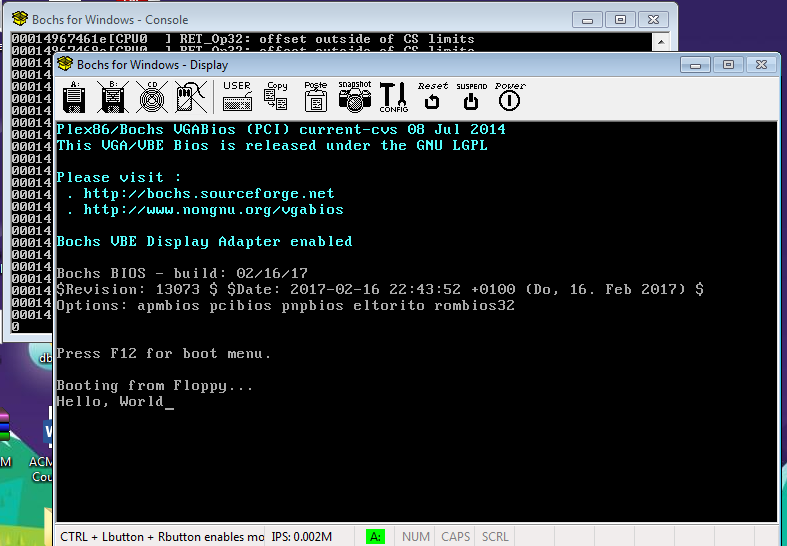
\_\_asm\_\_ \_\_volatile\_\_("movb $'d' , %al\n");

\_\_asm\_\_ \_\_volatile\_\_("movb $0x0e, %ah\n");

\_\_asm\_\_ \_\_volatile\_\_("int $0x10\n");

}

OUTPUT:



**MODULE 2:** Displaying rectangles as pixels

Test.c file:

\_\_asm\_\_(".code16gcc\n");

\_\_asm\_\_("jmpl $0x0000, $\_\_main\n");

#define MAX\_COLS 320

#define MAX\_ROWS 200

void printString(const char\* pStr) {

while(\*pStr) {

\_\_asm\_\_ \_\_volatile\_\_ (

"int $0x10" : : "a"(0x0e00 | \*pStr), "b"(0x0007)

);

++pStr;

}

}

void getch() {

\_\_asm\_\_ \_\_volatile\_\_ (

"xorw %ax, %ax\n"

"int $0x16\n"

);

}

void drawPixel(unsigned char color, int col, int row) {

\_\_asm\_\_ \_\_volatile\_\_ (

"int $0x10" : : "a"(0x0c00 | color), "c"(col), "d"(row)

);

}

void initEnvironment() {

\_\_asm\_\_ \_\_volatile\_\_ (

"int $0x10" : : "a"(0x03)

);

\_\_asm\_\_ \_\_volatile\_\_ (

"int $0x10" : : "a"(0x0013)

);

}

void initGraphics() {

int i = 0, j = 0;

int m = 0;

int cnt1 = 0, cnt2 =0;

unsigned char color = 10;

for(;;) {

if(m < (MAX\_ROWS - m)) {

++cnt1;

}

if(m < (MAX\_COLS - m - 3)) {

++cnt2;

}

if(cnt1 != cnt2) {

cnt1 = 0;

cnt2 = 0;

m = 0;

if(++color > 255) color= 0;

}

/\* (left, top) to (left, bottom) \*/

j = 0;

for(i = m; i < MAX\_ROWS - m; ++i) {

drawPixel(color, j+m, i);

}

/\* (left, bottom) to (right, bottom) \*/

for(j = m; j < MAX\_COLS - m; ++j) {

drawPixel(color, j, i);

}

/\* (right, bottom) to (right, top) \*/

for(i = MAX\_ROWS - m - 1 ; i >= m; --i) {

drawPixel(color, MAX\_COLS - m - 1, i);

}

/\* (right, top) to (left, top) \*/

for(j = MAX\_COLS - m - 1; j >= m; --j) {

drawPixel(color, j, m);

}

m += 6;

if(++color > 255) color = 0;

}

}

void \_main() {

printString("Now in bootloader...press any key\n\r");

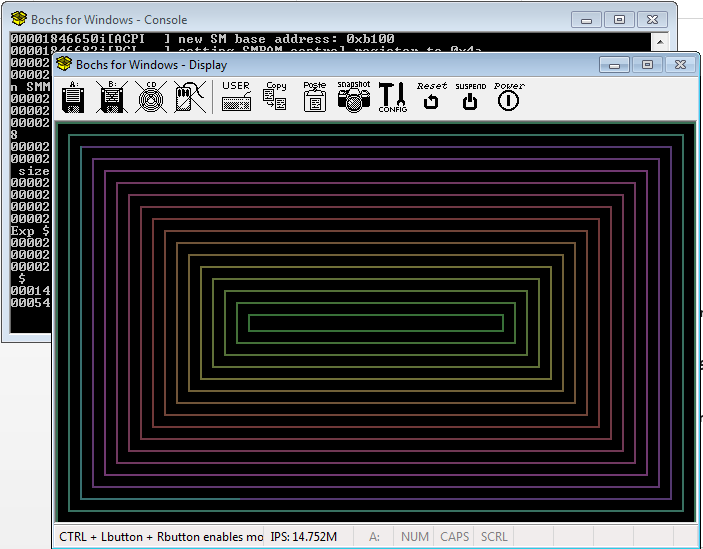
getch();

initEnvironment();

initGraphics();

}

**OUPUT:**



**LITERATURE REVIEW**

**[1] Cammack, W. E., & Kridner, J. D. (2007). *U.S. Patent No. 7,237,121*. Washington, DC: U.S. Patent and Trademark Office.**

**[2] Houha, J. A., Aspromonte, B. J., & Stalker, A. J. (2002). *U.S. Patent No. 6,393,585*. Washington, DC: U.S. Patent and Trademark Office.**

**[3] Yang, J., & Zhao, G. (2007). The transplant and compile of Bootloader in embedded system. *JOURNAL-SICHUAN UNIVERSITY NATURAL SCIENCE EDITION*, *44*(4), 835.**

**[4] LIU, H., ZOU, X. C., LI, P., LIU, M., & REN, H. Q. (2010). BootLoader Design for Terminal in Wireless Sensor Network [J]. *Computer Engineering*, *2*, 013.**

**[5] Triece, J., & Phoenix, T. (2001). *U.S. Patent Application No. 09/955,328*.**

**[6] Liu, Y., & Chen, Z. (2011). New Method for Flash Booting Based on TMS320C6000 DSP [J]. *Video Engineering*, *21*, 016.**

**[7] Yang, C. J., & Huang, M. J. (2010). *U.S. Patent Application No. 12/825,322*.**

**[8] Murray, M. R. (2009). *U.S. Patent Application No. 12/485,355*.**

**[9] JIE, Q., & GUOYING, L. (2007). The research of DSP booting from Flash [J]. *Microcomputer Information*, *32*, 080.**

**[10] Legat, U. (2008, October). Embedded system web server. In *9th International PhD Workshop on Systems and Control: Young Generation Viewpoint*.**