Contents

[1 Chapter 1 3](#_Toc6593173)

[1.1 Write and run a short program: *first* 3](#_Toc6593174)

[1.2 Concepts explanation: Terminal, assembler, linker and emulator 4](#_Toc6593175)

[1.2.1 Familiar yourself with Ubuntu desktop environment 4](#_Toc6593176)

[1.2.2 Terminal and CLI 4](#_Toc6593177)

[1.2.3 Binutils, as and ld. 4](#_Toc6593178)

[1.2.4 Qemu emulator 5](#_Toc6593179)

[1.3 Programs explanation: apt, as, ld and Qemu 5](#_Toc6593180)

[1.4 Concepts explanation: Von neumann architecture, bit, byte & legacy BIOS 6](#_Toc6593181)

[1.4.1 Von neumann architecture introduction 6](#_Toc6593182)

[1.4.2 Bit and Byte 7](#_Toc6593183)

[1.4.3 Memory is just a pile of boxes 8](#_Toc6593184)

[1.4.4 CPU is a black box with lots of registers 9](#_Toc6593185)

[1.4.5 I/O devices 9](#_Toc6593186)

[1.4.6 Binary and hexadecimal numbers 9](#_Toc6593187)

[1.4.7 Legacy BIOS 10](#_Toc6593188)

[1.5 Explanation of assembly source file first.s: Directives, instruction, label, interrupt, legacy BIOS attributes 10](#_Toc6593189)

[2 Chapter 2 14](#_Toc6593190)

[2.1 Memory map, stack & segment, video colour text memory area, function, ASCII code, suffix, x86 assembly language 14](#_Toc6593191)

[2.1.1 Memory map for "Low" memory (< 1 MiB) 14](#_Toc6593192)

[2.1.2 Real mode, memory addressing and high memory Area 15](#_Toc6593193)

[2.1.3 Stack & segment registers 16](#_Toc6593194)

[2.1.4 Real mode Flat model diagram 16](#_Toc6593195)

[2.1.5 Real mode segmented model 17](#_Toc6593196)

[2.1.6 Video colour text memory area 18](#_Toc6593197)

[2.1.7 x86 general purpose registers, x86 instruction reference and RFLAGS registers 19](#_Toc6593198)

[2.1.8 ASCII code 20](#_Toc6593199)

[2.2 Program ep0, write a single character to the video colour text memory 21](#_Toc6593200)

[2.3 Program ep0A, write 128 ASCII code to the video colour text memory 21](#_Toc6593201)

[Coming Chapters or Sections 24](#_Toc6593202)

[Appendix A Installing Ubuntu Desktop 24](#_Toc6593203)

[References 24](#_Toc6593204)

[Some useful references 24](#_Toc6593205)

# 1 Chapter 1

A Linux operating system is to be used as the platform for the experiments we are going to do in this book. Please move to section 1 if you currently have any distribution of Linux installed on your personal computer, it can be running on a virtual machine or alongside with Windows or macOS.

For these who do not currently have a Linux system in hand, or these who are even new to Linux system, Ubuntu desktop is recommended. Ubuntu is a free and open-source Linux distribution. It can be installed either on a virtual machine which is running on your current operating system or alongside with your current operating system. Please refer to appendix A for more information if you encounter some difficulties in installing Ubuntu desktop. It might take several hours if it’s your first time to install a system. Be patient and keep searching the answers whenever something confuses you.

## 1.1 Write and run a short program: *first*

The best way to learn is by doing. Please note I am using Ubuntu 18.04.2 LTS, the commands or operations can be different from these given in this book if you are using any other distribution of Linux. In this case, I suppose you know how to properly change the commands or operations as you are an experienced Linux user. Log into Ubuntu desktop and do the following steps:

1. Write the first program. Find the **Documents** folder and click into it. Create a text file and name it with **first.s**. Open it, copy and paste the following lines into the file then save and close.

.code16

.global \_start

\_start:

mov $26, %cx

mov $0x0903, %dx

mov $0x000c, %bx

mov $msg, %bp

mov $0x1301, %ax

int $0x10

loop: jmp loop

msg: .ascii "My first computer program!"

.org 510

.word 0xAA55

Congratulations! We have finished the hardest part of the whole chapter. Do not worry at the moment if you have no idea on what you have pasted, all these will be explained in the following sections. Please move to the next chapter, or skip any parts you’ve already known if you understand fully what the code does.

1. Open a terminal. Right click on the blank area inside of **Documents** folder and then click **Open Terminal** in the context menu.
2. Install **binutils** tool. Key in **sudo apt install binutils** and press **Enter** button. Input your password when asked.
3. Assemble the program: **first.s**. Inside of the terminal, input the following command and press Enter button:

**as -o first.o first.s**

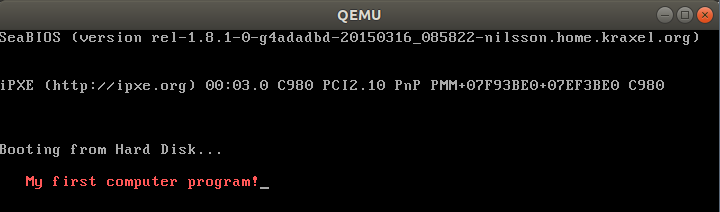
1. Generate the **first.img**. Input or copy and paste the following command and press **Enter** key:

**ld -Ttext=0x7c00 -o first.img --oformat=binary first.o**

1. Install **Qemu** emulator. Key in **sudo apt install Qemu**. Input **Y** and press **Entre** key when asked “Do you want to continue? [Y/n]”.
2. Run your program. Input or copy and paste the following command and press **Entre** key:

**sudo Qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img**

1. We will see the following window if everything has gone well so far. The light red words My first computer program! on the screen is what the program does.



1. We have finished first program and had it run. Again we mentioned several terms in the above steps, do not worry these for now, we will see the next sections for the explanations.

## 1.2 Concepts explanation: Terminal, assembler, linker and emulator

We go though some basic concepts before we explain what exactly we have done in the last section.

### 1.2.1 Familiar yourself with Ubuntu desktop environment

New to Linux? Give yourself half an hour to click each icon or button you can find in Ubuntu desktop to see what they are. Refer to the guide below or just search any topic you are interested in if necessary.

Ubuntu desktop guide: <https://help.ubuntu.com/stable/ubuntu-help/>

### 1.2.2 Terminal and CLI

On Linux desktop, the Terminal is a program where commands are used to interact with computer. Most people are familiar with GUI or graphical user interface. Instead of using mouse heavily to interact with all kinds of GUI items, we will use command-line interface or CLI to talk to computer. In case you are new to CLI, you may want to quickly go through chapter 2~4 of this book *The Linux Command Line* (<http://linuxcommand.org/tlcl.php>) written by Willian Shotts. Or search something like the most used 10 Linux commands and play with these commands for an hour. Both CLI and GUI are also called shell which in fact is a user interface for access to an operating system’s services. We call it a shell because it is the outermost layer outside of the most important part of the operating system: the kernel. Feel free to google any of these terms you are interested in while it’s good for now even it’s the first time we have heard any of these. We will have a more concrete understanding with time going on.

### 1.2.3 Binutils, as and ld.

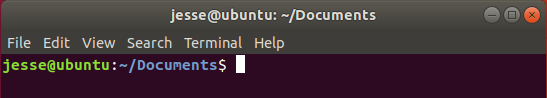
GNU Binutils are a collection of binary tools. The main ones are **as** and **ld**. The GNU assembler, commonly known as GAS or simply as. Assembler is a computer program which assembles assembly language to machine language. Assembly language is a more readable interpretation of a processor’s machine code, allowing easier understanding and programming by human [2]. Machine code is a computer program written in machine language [instructions](https://en.wikipedia.org/wiki/Instruction_set) that can be executed directly by a [computer](https://en.wikipedia.org/wiki/Computer)'s [central processing unit](https://en.wikipedia.org/wiki/Central_processing_unit) or CPU. Each instruction causes the CPU to perform a very specific task [3]. CPUs can only understand machine code. Machine code is some combination of statuses which only use two symbols: typically, “0” and “1”. The **ld** is another program, called linker, that takes one or more object files generated by the assembler and combines them into a single executable, library file, or another object file [4].

### 1.2.4 Qemu emulator

Qemu is a generic and open source machine emulator and virtualizer. We will see we can this emulator is another computer which runs on your host operating system, Ubuntu here. We will write some programs, put these programs into the virtual “hard disk” of this emulator and make the emulator to run our programs. Just like our real PC runs the operating system which sits on the real disk usually.

## 1.3 Programs explanation: apt, as, ld and Qemu

1. **In step 1 of section 1**, **we composed a text file using assembly language.** It’s OK for now if we do not really understand what the code means. We will examine these assembly code in the next sections and chapters. All we need to understand now is we wrote some assembly code and saved it as **first.s**, we call this file source file as it contains the source code. The suffix s here stands for source file.
2. **In step 2, we opened a terminal window.** The window will look like the below picture where **jesse** and **ubuntu** will be your user name and computer name. **~** indicate the current user’s home folder which is /home/*your\_user\_name*. Documents is just a folder under your home folder. We are now under Document folder because we right clicked in the blank area of this folder in the GUI. Believe you already tried to navigate to different folders following some books or web pages which tell the basic Linux commands usage. The place the cursor flashes is where we key in the commands to interact with the computer, just after the $ sign.



1. **Advanced package tool or apt** is a program that handles the installation and removal of software Ubuntu and related Linux distributions. **sudo** is short for *superuser do*, which is a program allows us to run programs with the security privileges of another user, by default the superuser. We use **sudo apt install binutils** when installing the tool collection binutils. Program **as** and **ld** which are part of the binutils will be used to translate the source file first.s to a runnable file or executable.
2. Let’s see the command in step 4 **as -o first.o first.s**. **as** is the program name of GNU assembler. Except for ‘--’ any command-line argument that begins with a hyphen (‘-’) is an option. The **-o** in **as -o first.o first.s** means we want to give the output file which is generated by the assembler the name **first.o**. Number of options can be zero or many. Anything that is not an option will be treated as a source file, like the **first.s** here. After the execution of this command, we get an object file **first.o** which is the input file of the program **ld**.
3. In Step 5, the option **-Ttext=0x7c00** tells the linker to locate the text section in the output file at the absolute address 0x7c00. 0x7c00 is a magic number, this is where the computer loads the data from the external storage like hard disk or USB flash drive into main memory. I understand this explanation might still confuse you. We will explain this in the following sections. Similarly, **-o first.img** tells the linker to generate the output file with the name **first.img**. Option **–oformat=binary** specifies the binary format for the output object file. Finally after the execution of this command we get the runnable file **first.img**. With this runnable file first.img, we would like to find a way to run it now.
4. We can image that the Qemu emulator, which has been installed in step 6, is our brand new machine which has exactly the function as the PC you buy from the shopping centre! But it’s not a real machine, it is only an emulator running on your Linux. We imagine it equips with one of the latest Intel or AMD CPUs (Intel and AMD are the companies who produce CPUs). The difference is the PC you buy from the shop comes with an operating system, usually Windows. While this machine does not come with any operating system. We will load the **first.img** file into the Qemu machine. We image the **first.img** file is equivalent to the files which are located on the hard disk of your real PC. I see this analogy might be not accurate enough from a computer scientist’ view, but I wish this is good enough for us to understand the relationship between the **first.img** file and the Qemu emulator for now.
5. The command in Step 7, **sudo Qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img**, **Qemu-sysem-x86\_64** is our brand new computer but without the operating system installed. Option **-cpu max** is used to enables all features supported by the accelerator in the current host machine. Option **-drive format=raw,file=first.img** is used to tell Linux to write the **first.img** into the disk of our brand new computer Qemu and then press the **Power on** button. Specifying **format=raw** avoids Qemu detecting the format and believe it’s a trusted format. **file=first.img** obviously tells Qemu which file to be loaded into the hard disk. You may have noticed the new term x86\_64, also known as x64, x86-64, AMD64 and Intel 64, is the 64-bit version of the x86 instruction set architecture. We have mentioned the smallest unit the CPU can execute is called an instruction. The set of all the instructions the CPU can understand is some kind of abstract model of a computer architecture. Almost all the PC and all kinds of Mac available in the shop in nowadays (this documents written in 2019) are using x86\_64 CPUs. Refer to the below link if you are interested in more information on x86\_64 and its family: <https://en.wikipedia.org/wiki/X86>
6. By now we have explained everything regarding the program and the commands we used so far but the most important part: what does the code in the text file mean. That’s the task for the following sections.

## 1.4 [Concepts explanation:](C:\\Users\\Jesse\\AppData\\C:\\Users\\Safemaster\\Desktop\\ttt\\exploring.docx) Von neumann architecture, bit, byte & legacy BIOS

Before we explain the code in the source file **first.s** line by line, a couple of concepts need to be introduced.

### 1.4.1 Von neumann architecture introduction

In 1945, a mathematician and physicist John von Neumann and some other people wrote a report which describes a design architecture for an electronic digital computer with these components:

1. A processing unit that contains an arithmetic unit and processor registers
2. A control unit that contains an instruction register and program counter
3. Memory that stores data and instructions
4. External mass storage
5. Input and output mechanisms[5]

Item iv correspondences to the disk or USB flash memory. The traditional hard disk which is an electromechanical device that uses magnetic storage to store and retrieve digital information using one or more rigid rapidly rotating disks. This kind of disk has been serviced the computer industry for more than half century. Since 1990s, a new kind of storage device that uses integrated circuit assemblies to store data appears. It is called solid-state drive or SSD or sometimes solid-state disk although they do not have physical disks. SSD is much faster and expensive than the HDD for hard disk drive. Luckily we do not need to understand too much of their working principles. All we need to know both of them can store data persistently even with power off. No matter what kind of information it is, the existing form on the disk is always a series of two kind of status. We use ‘0’ and ‘1’ to indicate these two kinds of status. Thus the information in a disk or USB drive or DVD disc is just a series of ‘0’s and ‘1’s. Can the ‘0’s and ‘1’s represent all kinds of information like music, movie, cartoons, texts, and all kinds of pictures?

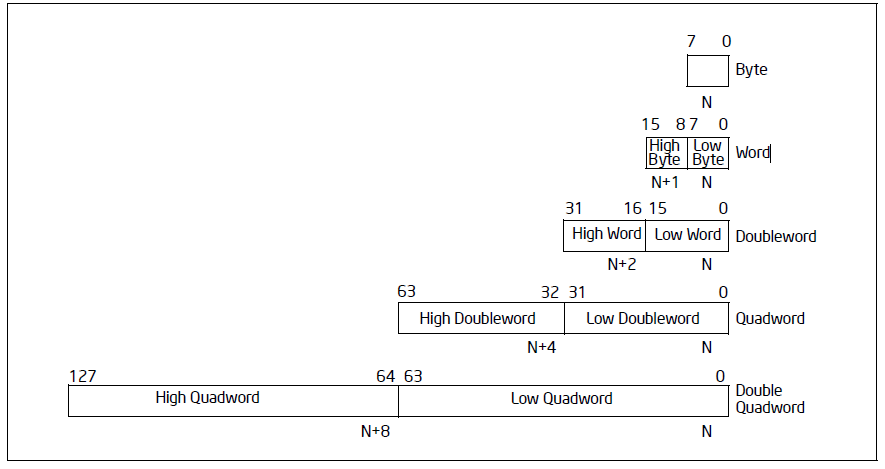
### 1.4.2 Bit and Byte

Now it’s a great time to introduce the concept of bit. We already know that the unit of information stored on a disk is either ‘1’ or ‘0’, logically we can imagine the data inside of a disk a combination of ‘0’s or ‘1’s. We call each of these smallest information unit a bit. The x86-64 architecture names a set of different data storage sizes as follows:

|  |  |  |
| --- | --- | --- |
| Storage | Size (bits) | Size (bytes) |
| Byte | 8-bit | 1 bytes |
| Word | 16-bit | 2 bytes |
| Double-word | 32-bit | 4 bytes |
| Quadword | 64-bit | 8 bytes |
| Double quadword | 128-bit | 16 bytes |

Table 1 Data type table

The below figure also gives the definition of Low Byte, High Byte, Low Word, High Word etc..



### 1.4.3 Memory is just a pile of boxes

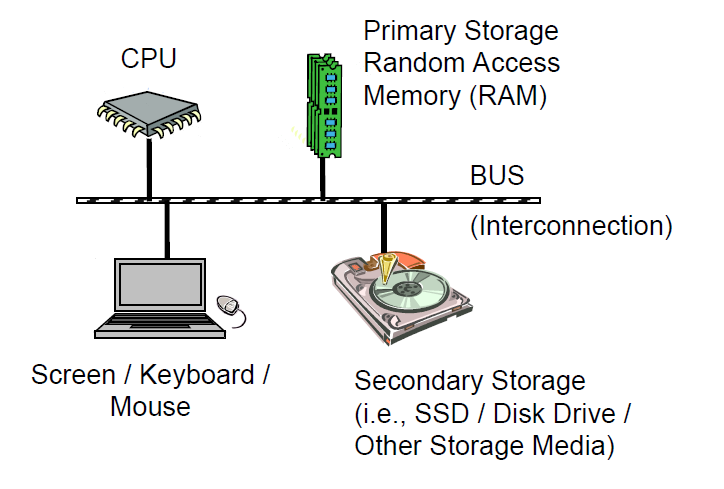
There are two main kinds of memory, volatile and non-volatile in a computer system. Non-volatile memory like ROM is used for storing firmware such as BIOS. We will talk about BIOS shortly. Firmware is a specific class of computer software that provides the low-level control for the device’s specific hardware. All we need to know for now is the BIOS system which is called firmware is stored in ROM, these data in ROM will not lost even when power off. The volatile memory are typically primary storage or main memory is random-access memory or RAM. We can imagine the RAM just like a piles of boxes which can store data. Inside of each box it’s either ‘0’ or ‘1’, each box represent a bit. When the computer system is running, CPU loads data from these boxes, the main memory, into CPU to do the computing. As there are so many boxes the CPU has to find a way to make sure it loads the right boxes, naturally the clever early computer scientists worked out a way: give these boxes an address. The minimum addressable unit of these boxes are 8, which is 8-bit or a byte. For now we can image that the main memory has a tall piles of boxes, each level contains 8 boxes, 1 byte or 8-bit. The lowest level byte comes with an address 0, the second lowest level byte with address 1, the third lowest level byte with address 2, and so on. And again these ‘0’s and ‘1’s in main memory disappear once power off.

### 1.4.4 CPU is a black box with lots of registers

The text book tells us there are two main components inside of a CPU, the control unit or processing unit. While for now we can think a CPU is just a black box, which means we do not need to understand how it works in the hardware level for now. All we need to know is it can execute instructions. We can image that a CPU is just like a mini robot. This little robot or just a black box each time reads in an instruction, it does something based on the instruction then next instruction will be fed to it. We already talked the minimum unit of command CPU can understand is an instruction. Each instruction itself is actually a number of ‘0’s and ‘1’s. Where are these instructions stored? On the surface of this little black box, there are many storage boxes, which are called registers. These registers reflect the CPU’s status, store the instructions and all kinds of data. We will use and learn more about the names and function of these registers.

### 1.4.5 I/O devices

We briefly talked about the CPU, main memory and disks, to conclude the first overview of the computer architecture, we have a look at the following illustration.

  
Illustration 1: Illustration: Computer Architecture[6]

The above illustration summaries well what have discussed so far. All the other devices we have not mentioned like monitor/screen, keyboard and mouse etc., plus the SSD or HDD and other secondary storage media are all call I/O devices. The reason why we put all these except the CPU and main memory/RAM into one category (the I/O devices) is the CPU treats all of these devices in a very similar way. The BUS is just like the high ways, that’s where the CPU, main memory, and I/O devices communicate through.

### 1.4.6 Binary and hexadecimal numbers

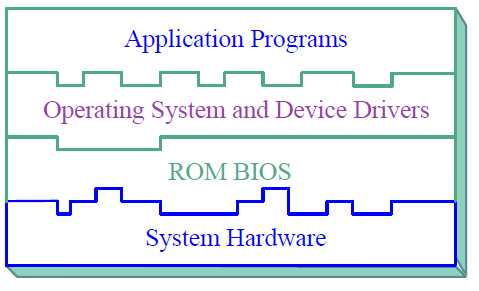
We learned that every kind of information stored in any kind of disks is just a number of ‘0’s and ‘1’s. Now we consider how to represent 0 or the positive integer numbers like 0, 1, 2, 3 etc. using these magic ‘0’s and ‘1’s. Say if we are just given two bits, we have four kinds of different combination of ‘0’s and ‘1’s: 00, 01, 10, 11. It’s very easy to think that we can just use binary values to represent integers. What is the biggest integer a byte can represent? 1111 1111, right? Some one might mistakenly reckon it’s a decimal number, so we use 0b prefix the number to indicate this is a binary number instead of a decimal number. So 0b11111111 is the biggest integer that 8 bits can represent. A bit hard to count the number of ‘1’s? Then hexadecimal numbers are used to make it much clearer, numbers 0-9 and letters a-f (or A-F) are used to represent decimal number 0 to15. So 0b1111 equals to 15 in decimal and 0xF in hexadecimal. Prefix 0x is used to indicate a hexadecimal. The biggest number a byte can represent which is 0xFF equals to 15x16+15 which is 255. Feel free to google and learn more about the binary and hexadecimal number until you feel comfortable. But do not worry too much we will have a deeper understanding when we read more or do more coding.

### 1.4.7 Legacy BIOS

A ROM BIOS (Basic Input/Output System) is a set of programs permanently stored in a ROM (Read-Only Memory) chip located on the computer motherboard. These programs micro-manage the hardware devices installed on the computer. When we turn on the computer, the ROM BIOS initializes and tests these devices. The first job of a ROM BIOS is to initialize and configure the computer hardware when we turn on the computer (system boot). The BIOS runs a series of complex programs called the Power On Self Test (POST), which performs a number of tasks, including: [8]

1. Test Random Access Memory (RAM)
2. Conduct an inventory of the hardware devices installed in the computer
3. Configure hard and floppy disks, keyboard, monitor, and serial and parallel ports
4. Configure other devices installed in the computer such as CD-ROM drives and sound cards
5. Initialize computer hardware required for computer features such as Plug and Play and Power Management
6. Run Setup if requested
7. Load and run the Operating System

The last task is to load the Operating System code usually from the disk. The executable file first.img is loaded into main memory and executed when the last command “sudo Qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img” is executed.[8]

  
Illustration 2: Illustration: Computer System Layers[8]

The above illustration explains well the layers for a computer system. ROM BIOS which is also called firmware sits in between the bare hardware and the Operating System. The application programs like a web browser, a word processing program, or a video player mount on the toppest layer.

## 1.5 Explanation of assembly source file first.s: Directives, instruction, label, interrupt, legacy BIOS attributes

Now let’s go through the assembly code we wrote in section 1.

1. Line 1 **.code16** is a directive which instructs the assembler, GNU as or GAS the one we use here, to generate the object file in 16-bit mode. As by default GAS generates 32-bit or 64-bit code depending on the configuration. The reason why we need 16-bit code is at the time when this program runs the CPU is in 16-bit real mode. All directives have names that begin with a period (‘.’) in GAS. These directives are not translated into any machine code but only tell GAS how to organise or translate the code into machine code.
2. The second line code **.global** is a directive as well. The function of this directive is to make the symbol, which is **\_start** in this program, visible to linker. Program **first.s** only has one source file, while for programs which have more than one source file, the **.global** directive will make symbol is also available to other source files. Back to the third line of the program, **\_start** indicates the default entry point. Linker later makes the runnable file starts from the first instruction which is just after the symbol **\_start**.
3. In line 4 **mov $26, %cx**, where **mov** is its opcode, $26 and %cx are its two operands. An opcode or operation code can have zero to two operands. 26 with a prefix $ indicates an immediate number 26 in decimal. A register name **cx** with prefix % indicates the content inside of the register. This instruction tells CPU to move the immediate decimal number 26 into the register %**cx**, which is a 16-bit register.
4. Similarly line 5 **mov $0x0903, %dx** moves the data which is a hexadecimal number 0903 to register dx. Line 6 **mov $0x000c, %bx** moves the data which is a hexadecimal number 000c to register bx.
5. We look at line 11 **msg: .ascii “my first computer program”**. **.ascii** which starts with ‘.’ is a directive, it tells the assembler to reserve space for a string or text, which is the following “my first computer program” here. The **msg** before the colon (‘:’) is called a label. With this label we can refer the address where the string is in this case.
6. Line 7 **mov $msg, %bp** moves the address for the string of “my first computer program” to register &=%**bp**. Similarly line 8 **mov $0x1301, %ax** moves number 0x1301 to register **%ax**.
7. The **int** opcode here is an interrupt, which force the CPU stop executing current task and start calling a routine (we understand a routine or a handler or a function is a just another piece of code stored somewhere else). For this case, the operands $0x10 which is decimal number 16 is given to CPU then CPU searches a table called interrupt vector table or IVT to find the location of the handler and then execute the code inside of the handler. Once the interrupt handler finishes, the CPU comes back to execute the next instruction which it leaves before the interrupt. The concept of Interrupt is not hard to understand, is it? Actually this kind of interrupt is usually called software interrupt. There are two other kinds of interrupts, one is external interrupt or hardware interrupt, the other one is called internal interrupt or exception interrupt. We will talk these two later.
8. Line 10 **loop: jmp loop** lets the CPU jumps to the label loop which again let the CPU jump to the same instruction. So it’s a dead cycle which will freeze the monitor, the CPU would keep doing this instruction until we close the emulator.
9. The last two lines are all directives. **.org 510** tells the assembler to put the next instruction or data from the 510th byte of the whole executable file. The last line **.word 0xAA55** will fill a word which is two bytes at the 510th and 511st byte location of the file. Actually these two numbers are magic numbers. When the BIOS program starts searching any bootable disk, it examine the first 512 bytes of the disk if it finished with 0xAA55, it thinks it’s a bootable disk and then BIOS program copies this 512 bytes into the main memory (put them byte by byte from the address 0x7c00) then CPU starts executing the program for the main memory address 0x7c00. In our example, that’s the runnable code generated from our first assembly source file first.s.
10. Now the only problem is why assign register %cx, %dx, %bx, %bp and %ax with numbers or address 26, 0x0903, 0x000c, $msg and 0x1301. In order to understand this, we need to find out the interfaces the legacy BIOS defines. In other words what numbers or parameters we need to assign to the registers before the interrupt instruction.

The following web pages give an easy way to check the meanings of these numbers.

Legacy BIOS Interrupt Vector Table: <http://www.ctyme.com/intr/int.htm>

Legacy BIOS colours attributes: <https://en.wikipedia.org/wiki/BIOS_color_attributes>

Spend some time to study the above web pages, then try change the 0x000c to 0x0002 (will change to green colour). Re-assemble, re-link and launch the Qemu to run the program to see the colour changes as desired or not. Actually instead of keying in 3 command lines we can connect the three commands with a semicolon(‘;’):

**as -o first.o first.s;ld -Ttext=0x7c00 -o first.img --oformat=binary first.o;sudo Qemu-system-x86\_64 -cpu max -drive format=raw,file=first.img**

1. To conclude the codes, we think in this way to explain these assembly lines again. The intension of the program is to write a line “My first computer program” to somewhere on the screen. This explains well why we have line 11. **.ascii** which is a directive tells the assembler to save the whole string “My first computer program!” to somewhere in the final executable. We image there’s a data section in the final executable, this string is put in that section as part of the executable. **.msg** which is a label here, this is necessary because we then can reference this label somewhere else in the source file let the assembler know we want to use the string information there. This label is actually an address. We’ve already mentioned the executable will eventually loaded into the memory before the first instruction of the executable loaded into the dedicated register from where the CPU gets instruction and executes.

Then we need to tell where on the screen we hope the string can be printed and the length of the string. The two bytes 0x0903 in line 5 tells the computer the place where the string to be printed. The higher byte 09 and lower byte 03 tell the row number and column number where to print the first letter of the string. Feel free to change any of these two numbers and re-run the program to see how these numbers control the position of the string. The lower byte 0x01 in %bx defines the colours. The higher four bits in the lower byte, which is 0 in our original code, defines the background colour of the character. If you have already studies the page I gave you (<https://en.wikipedia.org/wiki/BIOS_color_attributes>), you will find we can define 16 different colours (as shown in the below table). In our code, the higher 4 bits in 0x01 is 0, which means the background of the string will be black according to the below table. Feel free to change 0 to any numbers between 1 to F (F is a hexadecimal number which is 15 in decimal), re-assemble, re-link, and re-run the program to see the effects. By the way, 4-bit which is also half byte has 16 different (24=16) combination of these ‘0’s and ‘1’s. We use these 16 different statuses to represent 16 different colours, clever? I think so. That’s one way how we can understand every kind information (colours here) can be represent by these ‘0’s and ‘1’s, as long as we have enough long bits to put these ‘0’s and ‘1’s. Generally, if we are given n bit (where n is a positive integer), we can fill these bits with either ‘0’s or ‘1’s into these bit (boxes) to represent 2n different status. Get back to the code. The lower 4 bits in 0x01 here is 1 which let the computer print blue colour characters as we have seen. Change the lower 4 bits to a different number to see the effects?

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Dec | Hex | Binary | Colour | |
| 0 | 0 | 0000 | Black |  |
| 1 | 1 | 0001 | Blue |  |
| 2 | 2 | 0010 | Green |  |
| 3 | 3 | 0011 | Cyan |  |
| 4 | 4 | 0100 | Red |  |
| 5 | 5 | 0101 | Magenta |  |
| 6 | 6 | 0110 | Brown |  |
| 7 | 7 | 0111 | Light Grey |  |
| 8 | 8 | 1000 | Dark Grey |  |
| 9 | 9 | 1001 | Light Blue |  |
| 10 | A | 1010 | Light Green |  |
| 11 | B | 1011 | Light Cyan |  |
| 12 | C | 1100 | Light Red |  |
| 13 | D | 1101 | Light Magenta |  |
| 14 | E | 1110 | Yellow |  |
| 15 | F | 1111 | White |  |

1. In line 7, the address of the string, which is msg here, is assigned to register %bp. Then we raise an interrupt in line 9 **int $0x10**, the CPU then stop the current task (which is our current program ***first***), move to some code which was already prepared by the BIOS before our program **first** runs. CPU based on the number which is %0x10 to check a table (this table actually stored in the first 1024 bytes of the memory), which pretty similar the way we check this page (<http://www.ctyme.com/intr/int.htm>), now follow me to click the number 10 in the above page, then we see another long list. Next step is to check the number in %ax, which is 0x1301 in our program. So we find the link Int **10/AH=13h** (**h** here indicate 13 is a hexadecimal number) and click into this link. Read the page we eventually understand fully why we put those numbers into the four registers %ax, %bx, %cx, and %dx. We have to tell clearly the position, the string content, the colour attributes etc., to the interrupt program which is part of the BIOS in order let the service program (the BIOS program called here) know what to do.
2. I suppose we have examined every part of the whole assembly code we wrote in **first.s**.

# 2 Chapter 2

## 2.1 Memory map, stack & segment, video colour text memory area, function, ASCII code, suffix, x86 assembly language

In this section we are going to introduce some concepts then we start composing another piece of assembly code **ep0.s**.

### 2.1.1 Memory map for "Low" memory (< 1 MiB[[1]](#footnote-1))

We have already introduced that once we power on the computer, the legacy BIOS runs a series of complex programs called Power On Self Test (or POST), then the BIOS transfers control to the boot sector. At this time, the first megabyte of memory looks like this[[2]](#footnote-2):

|  |  |  |  |
| --- | --- | --- | --- |
| Start Address | Size | End Address | Name |
| 0x00000 | 1 KiB | 0x003FF | Interrupt Vector Table |
| 0x00400 | 0.25 KiB | 0x004FF | BIOS Data Area |
| 0x00500 | 29.75 KiB | 0x07BFF | Free Memory |
| 0x07C00 | 0.5 KiB | 0x07DFF | Boot Sector Code |
| 0x07E00 | 480.5 KiB | 0x7FFFF | Free Memory |
| 0x80000 | 128 KiB | 0x9FFFF | Extended BIOS Data Area |
| 0xA0000 | 64 KiB | 0xAFFFF | Graphics Video Memory |
| 0xB0000 | 32 KiB | 0xB7FFF | Monochrome Text Video Memory |
| 0xB8000 | 32 KiB | 0xBFFFF | Colour Text Video Memory |
| 0xC0000 | 256 KiB – 16 bytes | 0xFFFEF | ROM Code Memory |
| 0xFFFF0 | 16 bytes | 0xFFFFF | More BIOS Data |

Table 1 Legacy BIOS Memory Map

The first 1024 bytes (1KiB) stores the interrupt vector table, which has 256 records, each takes 4 bytes (32 bits). Each vector or record is a pointer (point to the address of a handler, or function or procedure) that tells the CPU the location where the code associated with the interrupt located.

Then the following 256 bytes (0.25 KiB, 0x400 to 0x4FF) is a BIOS data area.

From 0x0500 to 0x7BFF that’s 29.75 KiB is guaranteed for free use, usually called conventional memory.

From 0x7C00 to 0x7DFF there are 512 bytes which is the boot sector. That’s the RAM area where our first program is loaded into. We will still write some code and load them here for a while.

There are another 480.5 KiB for conventional use just before the 128 KiB Extended BIOS Data Area (from 0x80000 to 0x9FFFF). The EBDA is a variable-sized memory area (on different BIOSes). It is always immediately below 0xA0000 in memory if it exists. Also, it is guaranteed to be at most 128 KiB in size.[[3]](#footnote-3)

The next 384 KiB is is reserved for graphics video data, ROM data and some other BIOS data. Inside of this part, there are 32 KiB for Colour Text Video Memory. We write the character code and its attributes into this memory area, then these will be displayed on the screen. We will learn how to do this in next section.

We do not discuss the memory area above 1 MiB in this section.

### 2.1.2 Real mode, memory addressing and high memory Area

**Real Mode**: Real Mode is a simplistic 16-bit mode that is present on all x86 processors. Real Mode was the first x86 mode design and was used by many early operating systems before the birth of Protected Mode. For compatibility purposes, all x86 processors begin execution in Real Mode.[[4]](#footnote-4) We will play for a while under this real mode until we switch from Real mode to Protected Mode for whose detail will be explained more later.

**Memory Addressing**: You may already have a question why we only list the first megabyte when we discuss the memory map. That’s because we only can access about 1 MiB memory in Real mode.

Firstly to see, if we only have 16 bits to store the address of the memory, these 16 bits area stored in a register called IP, see below figure. How many bytes of memory can be addressed? That’s 216=65,536 bytes which is 64 KiB. If we use another register, also 16 bits, stored in a register called CS, then every time we calculate the address we use 16 times the CS then plus the IP to get the physical address. To be simple, we actually can address 220= 1,048,576 bytes which is 1 MiB. Usually we call CS: IP the logical address, the address calculated using CS times 16 +IP the physical address. The address space in Real mode segmented model runs from 0x00000 to 0xFFFFF.

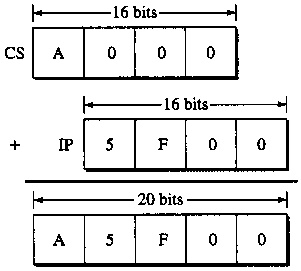


Figure 1Generating 20-bit physical address in Real Mode[[5]](#footnote-5)

**High memory Area**: Someone might argue that we can get address more than 0xFFFF, e.g., when CS=0xFFFF, IP=0xFFFF. We have to use the 21st address line to access any memory larger than 0xFFFFF. If we set segment register to a value of 0xFFFF, it points to an address that is 16 bytes below 1 MB. If we then use that segment register as a base, with an offset of 0x10 to 0xFFFF, we can access physical memory addresses from 0x100000 to 0x10FFEF. This (almost 64 KiB) area above 1 MiB is usually called the "High Memory Area" in Real Mode. Note that we have to have the A20 (the 21st) address line activated for this to work. We do not discuss more about A20 here. For now we are comfortable to limit our discussion on the 1 MiB address space in Real mode.

### 2.1.3 Stack & segment registers

All x86 segment registers are 16 bits in size, irrespective of the CPU:

* CS, code segment. Machine instructions exist at some offset into a code segment. The segment address of the code segment of the currently executing instruction is contained in CS.
* DS, data segment. Variables and other data exist at some offset into a data segment. There may be many data segments, but the CPU may only use one at a time, by placing the segment address of that segment in register DS.
* SS, stack segment. The stack is a very important component of the CPU used for temporary storage of data and addresses. Therefore, the stack has a segment address, which is contained in register SS.
* ES, extra segment. The extra segment is exactly that: a spare segment that may be used for specifying a location in memory.
* FS and GS are clones of ES, the extra segment. FS and GS both are just additional segments, no specialty here. Names FS and GS come from the fact that they were created after ES: E, F, G. They exist only in the 386 and later x86 CPUs.
* Extra segments ES, FS, and GS can be used for both data or code. 5

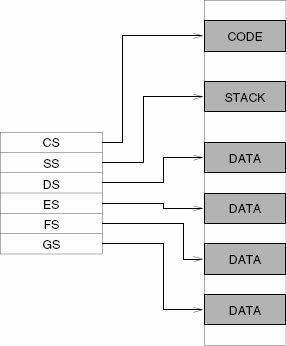


Figure 2six segments of the memory system5

### 2.1.4 Real mode Flat model diagram

* The segment registers are all set to point to the beginning of the 64 KiB block of memory.
* The operating system sets segment registers when it loads the program.
* All segment registers point to that same place.
* Physical segment assignments never change as long as the program is running.
* The segment registers are still functioning, but no work with segments is required.

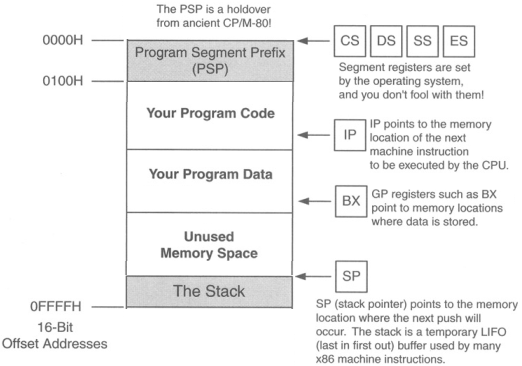


Figure 3Real mode Flat model diagram5

### 2.1.5 Real mode segmented model5

* Real mode segmented model was mainstream programming model throughout the MS-DOS era.
* Used when Windows 9x machine is booted into MS-DOS mode.
* Good choice to write code to run under MS-DOS.
* Program has access to 1MB of memory.
* The CPU handles transformations of segment:offset combinations into a full 20-bit address.
* CS always points to the current code segment

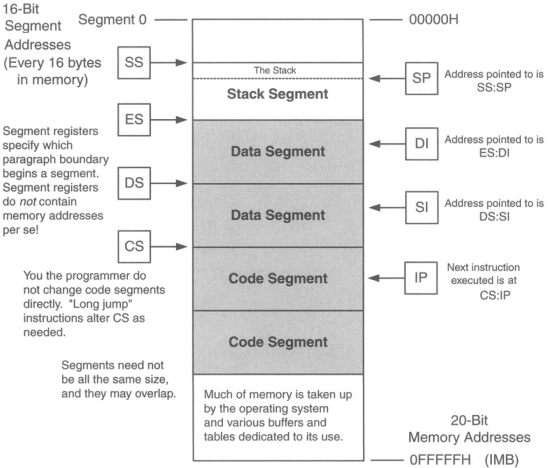


Figure 4Real mode segmented model5

* The next instruction to be executed is pointed to by the CS:IP register pair.
* Machine instructions called jumps can change CS to another code segment if necessary.
* The program can span several code segments.
* There is no direct CS manipulation to change from one code segment to another: when a jump instruction needs to take execution into a different code segment, it changes CS value for you.
* There is only one stack segment for any single program.
* A program has potential to destroy portions of memory that does not belong to its process.
* Careless use of segment registers will cause the operating system to crash.

### 2.1.6 Video colour text memory area

We actually print a string in our very first program first. We understand as long as we put the address of the string and the attributes like colour attributes and then raise the interrupt, the CPU will call BIOS procedure to print the string onto the string.

There are generally two ways to access VGA[[6]](#footnote-6) text-mode for an application: through the Video BIOS interface (which we have done in chapter 1) or by directly accessing video RAM and I/O ports. The latter method is considerably faster, and allows quick reading of the text buffer, for which reason it is preferred for advanced TUI[[7]](#footnote-7) programs.[[8]](#footnote-8)

The VGA text buffer is located at physical memory address 0xB8000. Since this is usually used by 16-bit x86 processes operating in real-mode, it is the first half of memory segment 0xB800. The text buffer data can be read and written, and bitwise operations can be applied. A part of text buffer memory above the scope of the current mode is accessible, but is not shown.8

Each screen character is actually represented by two bytes aligned as a 16-bit word accessible by the CPU in a single operation. The lower, or character, byte is the actual code point[[9]](#footnote-9) for the current character set, and the higher, or attribute, byte is a bit field used to select various video attributes such as colour, blinking, character set, and so forth.8

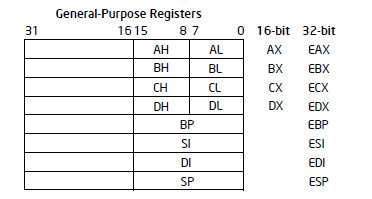
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Attribute (higher byte) | | | | | | | | Character (lower byte) | | | | | | | |
| 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Background colour | | | | Foreground colour | | | | Code point | | | | | | | |

Table 1Representation of colour text (Might be different based on the mode)

### 2.1.7 x86 general purpose registers, x86 instruction reference and RFLAGS registers

We’ve briefly discussed the segment registers. Now we introduce the general purpose registers before we do more code.

x86-64 has sixteen (almost) general purpose 64-bit integer registers. The above illustration shows the eight 32-bit general purpose register and their alternate names. In 64-bit mode there are another eight general purpose registers R8~R15, while we do not talk about these at the moment.

  
Illustration 1: 32-bit Alternate General-Purpose Register Names

Although the main registers (with the exception of the instruction pointer) are "general-purpose" in the 32-bit and 64-bit versions of the instruction set and can be used for anything, it was originally envisioned that they be used for the following purposes:

* AL/AH/AX/EAX/RAX: Accumulator
* BL/BH/BX/EBX/RBX: Base index (for use with arrays)
* CL/CH/CX/ECX/RCX: Counter (for use with loops and strings)
* DL/DH/DX/EDX/RDX: Extend the precision of the accumulator (e.g. combine 32-bit EAX and EDX for 64-bit integer operations in 32-bit code)
* SI/ESI/RSI: Source index for string operations.
* DI/EDI/RDI: Destination index for string operations.
* SP/ESP/RSP: Stack pointer for top address of the stack.
* BP/EBP/RBP: Stack base pointer for holding the address of the current stack frame.
* IP/EIP/RIP: Instruction pointer. Holds the program counter, the address of next instruction.

The **FLAGS** [register](https://en.wikipedia.org/wiki/Processor_register) is the [status register](https://en.wikipedia.org/wiki/Status_register) in [Intel](https://en.wikipedia.org/wiki/Intel) [x86](https://en.wikipedia.org/wiki/X86) [microprocessors](https://en.wikipedia.org/wiki/Microprocessor) that contains the current state of the processor. This register is [16 bits](https://en.wikipedia.org/wiki/16-bit) wide. Its successors, the **EFLAGS** and **RFLAGS** registers, are [32 bits](https://en.wikipedia.org/wiki/32-bit) and [64 bits](https://en.wikipedia.org/wiki/64-bit) wide, respectively. The wider registers retain compatibility with their smaller predecessors.

|  |  |  |
| --- | --- | --- |
| **Bit(s)** | **Label** | **Description** |
| 0 | CF | Carry Flag |
| 1 | 1 | Reserved |
| 2 | PF | Parity Flag |
| 3 | 0 | Reserved |
| 4 | AF | Auxiliary Carry Flag |
| 5 | 0 | Reserved |
| 6 | ZF | Zero Flag |
| 7 | SF | Sign Flag |
| 8 | TF | Trap Flag |
| 9 | IF | Interrupt Enable Flag |
| 10 | DF | Direction Flag |
| 11 | OF | Overflow Flag |
| 12~13 | IOPL | I/O Privilege Level |
| 14 | NT | Nested Task |
| 15 | 0 | Reserved |
| 16 | RF | Resume Flag |
| 17 | VM | Virtual-8086 Mode |
| 18 | AC | Alignment Check / Access Control |
| 19 | VIF | Virtual Interrupt Flag |
| 20 | VIP | Virtual Interrupt Pending |
| 21 | ID | ID Flag |
| 22-63 | 0 | Reserved |

Table 1 RFLAGS Register

For a full list of x86 instruction reference, please refer to *x86 and amd64 instruction reference*[[10]](#footnote-10).

### 2.1.8 ASCII code

ASCII, abbreviated from American Standard Code for Information Interchange, is a [character encoding](https://en.wikipedia.org/wiki/Character_encoding) standard for electronic communication. ASCII codes represent text in computers, [telecommunications equipment](https://en.wikipedia.org/wiki/Telecommunications_equipment), and other devices. Most modern character-encoding schemes are based on ASCII, although they support many additional characters.[[11]](#footnote-11)

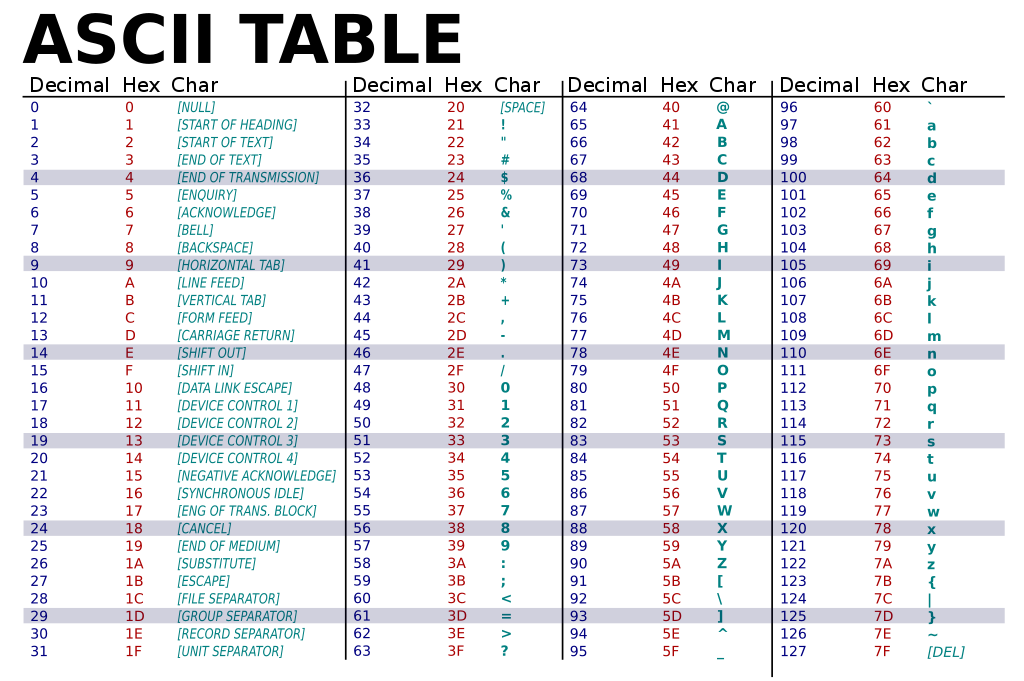


Figure 1ASCII Table

## 2.2 Program ep0, write a single character to the video colour text memory

In program ep0, we firstly set the segment registers %ds, %es and %ss to zero, then let %sp equal to 0x7c00, so the stack will grow downwards to the physical memory address 0.

Secondly, in the main program we assign the colour attribute and the ASCII code number, assign the address of the character position and the segment address of the colour text memory.

At last we call the function “write\_char” to write 2 bytes on to screen. Read the program and try to re-write your own version. Assembly and run the program will get the below output (the first character was overwritten by the character “C”).

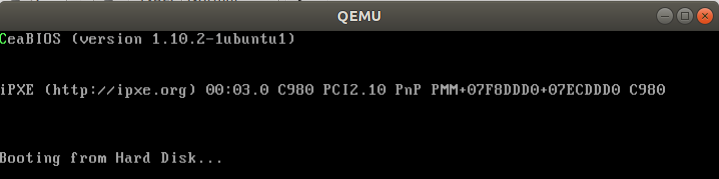


Figure 2 Result of program ep0

## 2.3 Program ep0A, write 128 ASCII code to the video colour text memory

In program ep0A, we will modify program ep0 to make it print the 128 ASCII code onto screen (33 of the 128 ASCII code are not printable, while different manufacturers extend these 33 code points differently. That’s why we can see some of these non-printable code points still be printed.).

We still call the function write\_char, then we set the %al to number 32 which is code point for space. Then we called write\_char 9 times to print 9 spaces before we write next code point (from 0 ~ 128). There will be 8 code points to be printed (as 80 character width in each row of the screen).

Inside of the write\_char function, code to check whether the current character position on screen added to reset the current character position to be zero when the last position is 1999.

Examine the code in ep0A.s and play with it. You will see the below output when you run it.

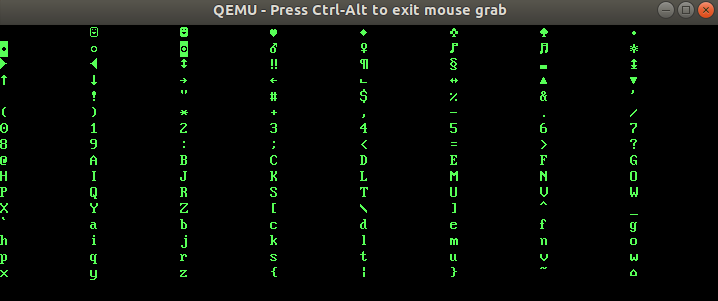


Figure 3Result of program ep0A

## 2.4 Extend program to ep1: add system\_interrupt

In this part, we firstly add a new function system\_interrupt, which for now just call the first function we wrote write\_char and then return (please note iret used instead of ret).

Then secondly we set the address of the function system\_interrupt to 0x0200. 0x0200 is the handler address of interrupt 0x80. As we already learnt the very first 1KiB in memory stores the Interrupt Vector Table[[12]](#footnote-12).

Lastly in the main part of the program, we use int $0x80 to raise an interrupt after we set the %al and %ah.

### 2.4.1 Interrupt Vector Table

In case some of us need to learn more about the Interrupt Vector Table. On the [x86](https://wiki.osdev.org/X86) architecture, the Interrupt Vector Table (IVT) is a table that specifies the addresses of all the 256 interrupt handlers used in [real mode](https://wiki.osdev.org/Real_mode).12

This website gives us a great idea on more details on the handers when an interrupt occurs. [[13]](#footnote-13)

## 2.5 ep2: add two functions task0 and task1

In this part we add two function, task0 to set a green colour, set code point to 67, then it raises an interrupt. This function will let the program keep printing character “C” in green to screen. The second function task1 will keep printing a magenta “S” to screen.

I found a little problem at first the program runs, it does not print a whole screen character as supposed. I go back to write\_char function and figure out we need to adjust the value of scn\_pos (plus 1 each time after we print 1 character).

===================

More links:

SEGMENT : OFFSET Addressing <https://thestarman.pcministry.com/asm/debug/Segments.html>

Bootstrapping: <http://www.read.seas.harvard.edu/~kohler/class/05f-osp/notes/lec03.html>

Computer Graphics: <http://www.brokenthorn.com/Resources/OSDevVga.html>

X86 registers: <http://liveforge.org/x86-registers-operating-modes/>

# 

# Coming Chapters or Sections

Real mode operating system.

## Appendix A Installing Ubuntu Desktop

You may want to install the Ubuntu on a virtual machine which is running on your current operating system, Windows or macOS. Go and search in your browser on how to install a virtual machine on your current operating system. For these who want to know what is a virtual machine or which virtual machine can be used, you may want to quickly review this page <https://en.wikipedia.org/wiki/Virtual_machine>.

For these who prefer to install the Ubuntu alongside with your current operating system or who have already installed a virtual machine, please move to Ubuntu official website to download the Ubuntu desktop. During the process of downloading, you may need search on how to install Ubuntu desktop on your virtual machine or alongside with your current operating system.

## References

[1] William E. Shotts, Jr., The Linux Command Line, Fifth Internet Edition, 2019, <http://linuxcommand.org/tlcl.php>, (accessed 3 April 2019).

[2] <https://en.wikipedia.org/wiki/Assembler>

[3] <https://en.wikipedia.org/wiki/Machine_code#Instruction_set>

[4] <https://en.wikipedia.org/wiki/Linker_(computing>[)](https://en.wikipedia.org/wiki/Linker_(computing))

[5] Von neumann architecture: <https://en.wikipedia.org/wiki/Von_Neumann_architecture>

[6] Ed Jorgensen, x86-64 Assembly Language Programming with Ubuntu, version 1.1.28, 2019, [www.egr.unlv.edu/~ed/assembly64.pdf](http://www.egr.unlv.edu/~ed/assembly64.pdf)

[7] Intel® 64 and IA-32 Architecture Software Developer’s Manual

[8] PhoenixBIOS 4.0 User Manual Revision 6

[9] Legacy BIOS Interrupt Vector Table: <http://www.ctyme.com/intr/int.htm>

[10] Legacy BIOS colours attributes: <https://en.wikipedia.org/wiki/BIOS_color_attributes>

[11] <https://en.wikipedia.org/wiki/Virtual_machine>

[12] <https://help.ubuntu.com/stable/ubuntu-help/>

[13] <https://en.wikipedia.org/wiki/X86>

[14] Memory Layout and Memory Map: <http://staff.ustc.edu.cn/~xyfeng/research/cos/resources/machine/mem.htm>

## Some useful references

GAS manual <https://sourceware.org/binutils/docs-2.32/as/index.html#SEC_Contents>

ld manual <https://sourceware.org/binutils/docs/ld/Options.html#Options>

Qemu manual

Introduction to x86 Assembly:

<https://software.intel.com/sites/default/files/m/d/4/1/d/8/Introduction_to_x64_Assembly.pdf>

x86 assembly reference: [www.felixcloutier.com/x86/](http://www.felixcloutier.com/x86/)

online x86 & x64 assembler and disassembler: <https://defuse.ca/online-x86-assembler.htm>

x86 assembly wiki book: <https://en.wikibooks.org/wiki/X86_Assembly>

x86 assembly with Ubuntu: [www.egr.unlv.edu/~ed/assembly64.pdf](http://www.egr.unlv.edu/~ed/assembly64.pdf)

using as, The GNU Assembler: <https://web.eecs.umich.edu/~prabal/teaching/resources/eecs373/Assembler.pdf>

<http://www.phoenix.com/resources/specs-bbs101.pdf>

<https://firmware.intel.com/sites/default/files/resources/A_Tour_Beyond_BIOS_Memory_Map_in%20UEFI_BIOS.pdf>

<https://www.cs.cmu.edu/~410/doc/minimal_boot.pdf>

<https://www.csie.ntu.edu.tw/~cyy/courses/assembly/10fall/lectures/handouts/lec12_x86arch.pdf>

1. What is a MiB or KiB: <https://en.wikipedia.org/wiki/Kibibyte> [↑](#footnote-ref-1)
2. Memory Layout and Memory Map: <http://staff.ustc.edu.cn/~xyfeng/research/cos/resources/machine/mem.htm> [↑](#footnote-ref-2)
3. Memory Map x86: <https://wiki.osdev.org/Memory_Map_(x86)#.22Low.22_memory_.28.3C_1_MiB.29> [↑](#footnote-ref-3)
4. Real Mode: <https://wiki.osdev.org/Real_Mode> [↑](#footnote-ref-4)
5. Modes of Memory Addressing on x86 <http://www.cjump.com/CIS77/ASM/Memory/lecture.html#M77_0020_seg_reg> [↑](#footnote-ref-5)
6. Video graphics array: <https://en.wikipedia.org/wiki/Video_Graphics_Array> [↑](#footnote-ref-6)
7. Text-based user interface: <https://en.wikipedia.org/wiki/Text-based_user_interface> [↑](#footnote-ref-7)
8. VGA compatible text mode: <https://en.wikipedia.org/wiki/VGA-compatible_text_mode> [↑](#footnote-ref-8)
9. Code point: <https://en.wikipedia.org/wiki/Code_point> [↑](#footnote-ref-9)
10. x86 and amd64 instruction reference: <https://www.felixcloutier.com/x86/> [↑](#footnote-ref-10)
11. ASCII: <https://en.wikipedia.org/wiki/ASCII> [↑](#footnote-ref-11)
12. Interrupt Vector Table: <https://wiki.osdev.org/Interrupt_Vector_Table#Structure> [↑](#footnote-ref-12)
13. Legacy BIOS Interrupt Vector Table: <http://www.ctyme.com/intr/int.htm> [↑](#footnote-ref-13)