**Chapter 1.Introduction**

**1.1 Required Knowledge**

Writing an OS is *not* a beginner's task. In fact, writing an OS is usually considered the most difficult programming task. You will need above-average programming skills before even considering a project like this. Failure to comply will make you look silly.

Some things you *will* need to know are:

* **Basic Computer Science**: You need to be intimately familiar with hexadecimal and binary notation as well as boolean logic and similar fundamental knowledge, like data structures, their construction and manipulation, searching and sorting algorithms, abstract programming concepts, etc. etc..
* **Language and Vocabulary**: You need to be able to read and write (technical) English at a competent level. Virtually all technical documentation is in English, and most of the resources you could find on the web are, also. Using incorrect terminology will make you look foolish and confuse the people willing to help you.
* **Language and Vocabulary, pt. 2**: Most operating systems are written in C(or C++). Even if you choose to use another language (like Pascal), C is the *lingua franca* of programming, and you should be competent in making heads and tails of it.
* **Assembly**: You should have knowledge about the low-level language Assembly Read a book. Take a course at school. Write some user-space code to familiarize yourself with it. You *will* need it, even if you plan to write most of your operating system in a higher-level language.
* **Programming experience**: Learning about programming with an OS project is considered a bad idea. Not only should you know the language in which you will be developing inside out, you should also be familiar with version control, debugging, etc. - in short, you should have written quite a few user-space programs in that language successfully before trying OS development.
* **UNIX experience**: You will soon notice that many of the tools used in OS development are developed for Unix, and later ported over to Windows. The Linux kernel is often used as reference or example for how things can be done, and many of the hobby operating systems have some resemblance to Unix. Having experience with the Unix command line (preferably bash or ksh) is a very important requirement. (Cygwin provides an easy-to-install Unix command line for Windows.). If you haven't, go ahead and use Linux or a BSD for a while.
* **Toolchain**: You must know the behavioral details of your compiler, assembler, linker, and make utility. You should have the documentation of the tools you use at hand, and refer to them *before* asking the community. Rest assured that any possible beginner's question about GCC, GNU as, NASM, GNU ld, Visual Studio and Grub has been answered twice over.
* **Emulators and Virtualizers**: Familiarity with tools such as Bochs, VirtualBox, QEMU, or VirtualPC is key to having a reasonable turn-around in development, and provide a buffer between your real hardware and your test system. While these can be learned for the specific purpose of OS dev, you will certainly want to be aware of what they are and where to get them before beginning an OS project.
* **Executable Formats**: Kernel space programming has many *additional* requirements unknown to application development. Being able to parse executable binaries is one of them (you *do* want your OS to load and execute applications, do you?) Make yourself familiar with Executable file types, their internal structure, and how a linker generates them.
* **The Platform**: You should have studied the manuals for the processor you will be programming for. They contain the information you need to design your kernel in the first place.
* **The Concept**: You should have an understanding of how existent operating systems function at the technical level (e.g. by having read some Books), what is good or bad about them, and how you want to go about your own OS. Having finished some tutorial and then asking "what now?" in one of the forums will just make you look silly.

**1.2 GCC Cross Compiler**

Generally speaking, a cross-compiler is a compiler that runs on platform A (the **host**), but generates executables for platform B (the **target**). These two platforms may (but do not need to) differ in CPU, operating system, and/or executable format. In our case, the host platform is your current operating system, and the target platform is the operating system you are about to make. It is important to realize that these two platforms are not the same; your operating system is always going to be different from your current operating system. This is why we need to build a cross-compiler first, you will most certainly run into trouble otherwise.

**Why do I need a Cross Compiler?**

* You need to use a cross-compiler *unless* you are developing on your own operating system. The compiler must know the correct target platform (CPU, operating system), otherwise you will run into trouble. If you use the compiler that comes with your system, then the compiler won't know it is compiling something else entirely. Some tutorials suggest using your system compiler and passing a lot of problematic options to the compiler. This will certainly give you a lot of problems in the future and the solution is build a cross-compiler. If you have already attempted to make an operating system without using a cross-compiler

**Which compiler version do I want?**

* The newest GCC is recommended as it is the latest and greatest release. However, it is recommended that you use the same major compiler version to build your cross-compiler. For instance, you may run into trouble if you use gcc 4.6.3 to build a gcc 4.8.0 cross-compiler. If you are not using the latest major GCC release for your system compiler, we recommend that you build the newest as your GCC compiler.
* You can also use older releases as they are usually reasonably good. If your local system compiler isn't too terribly old (at least gcc 4.6.0), you may wish to save yourself the trouble and just pick the latest minor release (such as 4.6.3 if your system compiler is 4.6.1) for your cross-compiler.

**1.3 Object files**

Object files basically consist of compiled and assembled code, data, and all the additional information necessary to make their content usable. In the process of building an operating system, you will use a lot of object files. While for common development tasks you do not need to know their exact details, when you want to create or use one with various specifics, the details can be very important.

## Objects and executables

Whereas wikipedia considers executables to be a subset of object files, there are significant differences. In some systems, they are a completely different format (COFF vs PE), or they have different fields (ELF program/section headers). The key difference is that in executables the addresses have been resolved, while in object files they have not. This means that non-executable files do not contain working code.

**Relocating code**

When an executable is created, it will be set to use a specific address by default. This can be a problem when you need several object files in the same address space and they may overlap, or you want to perform address space randomization, you might find relocating an executable an option.

Since relocations are only needed to build an executable, but not when you run it, they normally aren't present in a linked file. Instead you need to specifically tell the linker to emit relocations when necessary. For the GCC cross compiler, this can be done with the -q switch. Note that the -i and -r switches have a similar description, but cause the linker to yield an object file rather than an executable.

Relocating is of itself fairly straightforward by finding the differences. Start with loading the sections to the location of your choice, then for each relocation entry:

* compute the original address where the relocation was applied
* compute the address where the relocation applies now (its moved by the same amount you moved the original section from its original location)
* do the same for the destination of the relocation
* compute what the relocation value is - the destination for absolute relocations, and the destination minus the origin for relative relocations.
* compute what the relocation value was using the original location.
* subtract the old value from the new value
* add the result to the original relocation value in memory.

If the sections are moved relatively to each other, then relocating can become as simple as only adding the displacement to the absolute relocations. The relative locations do not get changed as both the source and the target are moved by the same amount.

**1.4 Which platform to use for development**

Do I already have to have an OS to make an OS?

Yes , Most people use Windows and/or Linux for OS development.

What compiler should I use?

For assembly language you need NASM(open source).available on both windows as well as linux.

For c/c++ gcc on Linux and DJGPP under Windows.

**1.5 Required tools**

* 2 PC (2nd can be Virtual.)
* Compiler

Linux: GCC

Windows: DJGPP

ALT: MS VISUAL STUDIO

* Editor:

Text Editor:

Linux: Gedit

Windows Notepad 2

IDE:

Eclipse , NetBeans

Win: MS VISUAL STUDIO

* Assembler

NASM

ALT:GAS

* Floppy

Real Drive ,OR:

Linux: DD and MOUNT

WIN: Virtual Floppy Drive

* Debugger

Bochs

* Emulator

Linux: QEMU , Virtual Box

WIM:MS Virtual PC 2007,Virtual Box,VMware

* Other tools:

Hex Viewer

Win:DEBUG

**1.6 Abstract flow chart of project**

**Chapter 2. Bootloader**

2.1 Booting Process

2.1.1Pressing the power button

What actually happens when you press the power button? When this button is pressed, the wires connected to the button send an electronic signal to the motherboard. The motherboard simply reroutes this signal to the power supply (PSU).

This signal contains a single bit of data. If it is 0, there is, of course, no power (so the computer is off, or the motherboard is dead). If it is a 1 (meaning an active signal), it means that power is being supplied.

To better understand this, remember the basics of binary logic in computers. 8 "bits" simply represent 8 "wires" or "lines" where electricity can go. A 0 represents no current, while a 1 represents current within a line. This, along with Logic Gates, is the bases of Digital Logic Electronics, at which computers were built.

When the PSU recieves this active signal, it begins supplying power to the rest of the system. When the correct amount of power is supplied to all devices, the PSU will be able to continue suppling that power without any major problems.

The PSU then sends a signal, called the "power\_good" signal into the motherboard to the Basic Input Output System (BIOS).

2.1.2 BIOS POST

when the BIOS recieves this "power\_good" signal, the BIOS begins initializing a process called POST (Power On Self Test). The POST then tests to insure there is good amount of power being supplied, the devices installed (such as keyboard, mouse, USB, serial ports, etc.), and insures the memory is good (By testing for memory curruption).

The POST then gives control to the BIOS. The POST loads the BIOS at the end of memory (Might be 0xFFFFF0) and puts a jump instruction at the first byte in memory.

The processors Instruction Pointer (CS:IP) is set to 0, and the processor takes control.

What does this mean? The processor starts executing instructions at address 0x0. In this case, it is the jump instruction placed by the POST. This jump instruction jumps to 0xFFFFF0 (or wherever the BIOS was loaded), and the processor starts executing the BIOS.

The BIOS takes control...

2.1.3 The BIOS

The Basic Input Output System (BIOS) does several things. It creates an Interrupt Vector Table (IVT), and provides some basic interrupt services. The BIOS then does some more tests to insure there is no hardware problems. The BIOS also supplies a Setup utility.

The BIOS then needs to find an OS. Based on the boot order that you set in the BIOS Setup, the BIOS will execute Interrupt (INT) 0x19 to attempt to find a bootable device.

If no bootable device is found (INT 0x19 returns), the BIOS goes on to the next device listed in the boot order. If there is no more devices, it will print an error simular to "No Operating System found" and halt the system.

2.2 Bootloader Theory

A device is bootable if it carries a boot sector with the byte sequence 0x55, 0xAA in bytes 511 and 512 respectively. When the BIOS finds such a boot sector, it is loaded into memory at 0x0000:0x7c00(segment 0, address 0x7c00). (However, some BIOS' load to 0x7c0:0x0000 (segment 0x07c0, offset 0), which resolves to the same physical address, but can be surprising. A good practice is to enforce CS:IP at the very start of your boot sector.)

Execution is then transferred to the freshly loaded boot record. On a floppy disk, all 512 bytes of the boot record may contain executable code. On a hard drive, the Master Boot Record (MBR) holds executable code at offset 0x0000 - 0x01bd, followed by table entries for the four primary partitions, using sixteen bytes per entry (0x01be - 0x01fd), and the two-byte signature (0x01fe - 0x01ff).

So far, bootloaders...

* ...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.

Simplest BootLoader:

org 0x7c00 ;We are loaded by BIOS at 0x7c00

bits 16 ;We are still in 16 bit Real Mode

Start:

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

**Chapter 3. Switching To Protected Mode**

**3.1 Basics of protected mode**

Protected Mode (PMode) is an operation mode available from the 80286 and later processors. PMode was primarily designed to increase the stability of the systems. As you know from the previous tutorials, Real Mode has some big problems. For one, we can write a byte anywhere we want. This can overwrite code or data, that may be used by software ports, the processor, or even our self. And yet, we can do this in over 4,000 different ways--both directly and indirectly!

Real Mode has no **Memory Protection**. All data and code are dumped into a single all purpose use memory block.

In Real Mode, you are limited to 16 bit registers. Because of this, you are limited to 1 MB of memory.

No support for hardware level **Memory Protection** or **Multitasking**.

Quite possibly the biggest problem, was that there is no such thing as "rings". All programs execute at Ring 0 level, as every program has full control over the system. This means, in a single tasking environment, a single instruction (such as **cli/hlt**) can crash the entire OS if you are not carefull.

A lot of this should sound familiar from when we covered Real Mode in depth. Protected Mode fixes all of these problems.

Protected Mode:

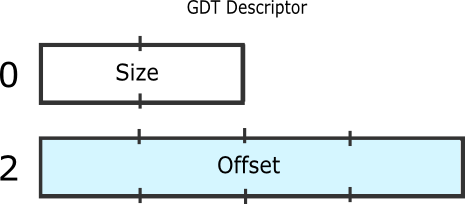
* Has **Memory Protection**
* Has hardware support for **Virtual Memory** and **Task State Switching (TSS)**
* Hardware support for interrupting programs and executing another
* 4 Operating Modes: **Ring 0, Ring 1, Ring 2, Ring 3**
* Access to 32 bit registers
* Access to up to 4 GB of memory

Remember that **we are in Ring 0, while normal applications are in Ring 3 (Usually).** We have access to special instructions and registers that normal applications do not. In this tutorial, we are going to be using the **LGDT** instruction, along with a **far jump** using our own defined segment, and the use of the **processor control registers.** None of this is available in normal programs.

|  |  |  |  |
| --- | --- | --- | --- |
| **Start** | **End** | **Size** | **Description** |
| 0x00000000 | 0x000003FF | 1kb | Interrupt vector table |
| 0x00000400 | 0x000004FF | 256bytes | BIOS data |
| 0x00000500 | 0x00007BFF | Almost 30kb | Conventional Memory |
| 0x00007C00 | 0x00007DFF | 512bytes | OS Boot sector |
| 0x00007E00 | 0x0007FFFF | 480.5kb | Conventional Memory |
| 0x00080000 | 0x0009FBFF | 120kb | Conventional Memory |
| 0x0009FC00 | 0x0009FFFF | 1kb | Extended BIOS data area |
| 0x000A0000 | 0x000FFFFF | 384kb | Video Memory |
| 0x00100000 | Above | ------------ | Free Memory |

Programs

**3.2 Global Descriptor Table(GDT)**



The **Global Descriptor Table** (**GDT**) is specific to the IA32 architecture. It contains entries telling the CPU about memory segments. A similar Interrupts Descriptor Table exists containing tasks and interrupts descriptors. The GDT is loaded using the LGDT assembly instruction.

The offset is the linear address of the table itself, which means that paging applies. The size is the size of the table subtracted by 1. This is because the maximum value of size is 65535, while the GDT can be up to 65536 bytes (a maximum of 8192 entries). Further no GDT can have a size of 0.

The table contains 8-byte entries. Each entry has a complex structure:

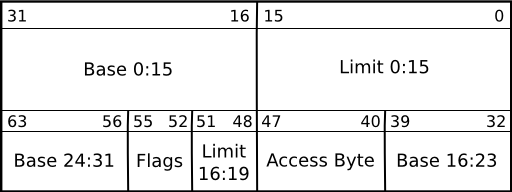


Fig.GDT Entry

What "Limit 0:15" means is that the field contains bits 0-15 of the limit value. The base is a 32 bit value containing the linear address where the segment begins. The limit, a 20 bit value, tells the maximum addressable unit (either in 1 byte units, or in pages). Hence, if you choose page granularity (4 KiB) and set the limit value to 0xFFFFF the segment will span the full 4 GiB address space. Here is the structure of the access byte and flags:



Fig.GDT Bits

The bit fields are:

* **Pr:** Present bit. This must be **1** for all valid selectors.
* **Privl:** Privilege, 2 bits. Contains the [ring level](http://wiki.osdev.org/Security#Rings), 0 = highest (kernel), 3 = lowest (user applications).
* **Ex:** Executable bit. If **1** code in this segment can be executed, ie. a code selector. If **0** it is a data selector.
* **DC:** Direction bit/Conforming bit.
  + Direction bit for data selectors: Tells the direction. **0** the segment grows up. **1** the segment [grows down](http://wiki.osdev.org/Expand_Down), ie. the offset has to be greater than the limit.
  + Conforming bit for code selectors:
    - If **1** code in this segment can be executed from an equal or lower privilege level. For example, code in ring 3 can far-jump to *conforming* code in a ring 2 segment. The privl-bits represent the highest privilege level that is allowed to execute the segment. For example, code in ring 0 cannot far-jump to a conforming code segment with privl==0x2, while code in ring 2 and 3 can. Note that the privilege level remains the same, ie. a far-jump form ring 3 to a privl==2-segment remains in ring 3 after the jump.
    - If **0** code in this segment can only be executed from the ring set in privl.
* **RW:** Readable bit/Writable bit.
  + Readable bit for code selectors: Whether read access for this segment is allowed. Write access is never allowed for code segments.
  + Writable bit for data selectors: Whether write access for this segment is allowed. Read access is always allowed for data segments.
* **Ac:** Accessed bit. Just set to **0**. The CPU sets this to **1** when the segment is accessed.
* **Gr:** Granularity bit. If **0** the limit is in 1 B blocks (byte granularity), if **1** the limit is in 4 KiB blocks (page granularity).
* **Sz:** Size bit. If **0** the selector defines 16 bit protected mode. If **1** it defines 32 bit protected mode. You can have both 16 bit and 32 bit selectors at once.

Not shown in the picture is the 'L' bit (bit 21, next to 'Sz') which is used for x86-64 mode.

**Chapter 4. Way To Interrupts**

**Interrupt Descriptor table**

The **Interrupt Descriptor Table** (**IDT**) is specific to the IA-32 architecture. It is the Protected mode counterpart to the Real Mode Interrupt Vector Table (IVT) telling where the Interrupt Service Routines(ISR) are located. It is similar to the Global Descriptor Table in structure.

The IDT entries are called gates. It can contain Interrupt Gates, Task Gates and Trap Gates.

Location of IDT (address and size) is kept in the IDTR register of the CPU, which can be loaded/stored using LIDT, SIDT instructions.

|  |  |  |
| --- | --- | --- |
| Name | Bit | Description |
| Limit | 0..15 | Defines the length of the IDT in bytes - 1 (minimum value is 100h, a value of 1000h means 200h interrupts). |
| Base | 16..47 | This 32 bits are the linear address where the IDT starts (INT 0) |

Fig. IDTR

This is similar to the [GDT](http://wiki.osdev.org/GDT), except:

* The first entry (at zero offset) is used in the IDT.
* There are 256 interrupts (0..255), so IDT should have 256 entries, each entry corresponding to a specific interrupt.
* It can contain more or less than 256 entries. More entries are ignored. When an interrupt or exception is invoked whose entry is not present, a GPF is raised that tells the number of the missing IDT entry, and even whether it was hardware or software interrupt. There should therefore be at least enough entries so a GPF can be caught.

The table contains 8-byte Gate entries. Each entry has a complex structure:

struct IDTDescr{

uint16\_t offset\_1; *// offset bits 0..15*

uint16\_t selector; *// a code segment selector in GDT or LDT*

uint8\_t zero; *// unused, set to 0*

uint8\_t type\_attr; *// type and attributes, see below*

uint16\_t offset\_2; *// offset bits 16..31*

};

The offset is a 32 bit value, split in two parts. The selector is a 16 bit value and must point to a valid selector in your GDT.

type\_attr is specified here:

7 0

+---+---+---+---+---+---+---+---+

| P | DPL | S | GateType |

+---+---+---+---+---+---+---+---+

**4.2 Progtrammable interrupt controller**

here are two types of interrupts, those generated by software (Useually by an instruction, such as INT, INT 3, BOUND, INTO), and an interrupt generated by hardware.

Hardware interrupts are very important for PC's. It allows other hardware devices to signal the CPU that something is about to happen. For example, a keystroke on the keyboard, or a single clock tick on the internal timer, for example.

We will need to map what Interrupt Request (IRQ) to generate when these interrupts happen. This way, we have a way to track these hardware changes.

Lets take a look at these hardware interrupts.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **x86 Hardware Interrupts** | | | | |
| 8259A Input pin | Interrupt Number | Description |  |  |
| IRQ0 | 0x08 | Timer | | | |
| IRQ1 | 0x09 | Keyboard | | | |
| IRQ2 | 0x0A | Cascade for 8259A Slave controller | | | |
| IRQ3 | 0x0B | Serial port 2 | | | |
| IRQ4 | 0x0C | Serial port 1 | | | |
| IRQ5 | 0x0D | AT systems: Parallel Port 2. PS/2 systems: reserved | | | |
| IRQ6 | 0x0E | Diskette drive | | | |
| IRQ7 | 0x0F | Parallel Port 1 | | | |
| IRQ8/IRQ0 | 0x70 | CMOS Real time clock | | | |
| IRQ9/IRQ1 | 0x71 | CGA vertical retrace | | | |
| IRQ10/IRQ2 | 0x72 | Reserved | | | |
| IRQ11/IRQ3 | 0x73 | Reserved | | | |
| IRQ12/IRQ4 | 0x74 | AT systems: reserved. PS/2: auxiliary device | | | |
| IRQ13/IRQ5 | 0x75 | FPU | | | |
| IRQ14/IRQ6 | 0x76 | Hard disk controller | | | |
| IRQ15/IRQ7 | 0x77 | Reserved | | | |

We will be able to install our own interrupt handlers within the **Interrupt Descriptor Table (IDT)** very easily. We create interrupt handlers to handle not only software interrupts, but interrupts triggered by hardware devices. **Remember: The hardware devices signal the Programmable Interrupt Controller to signal the processor to request a hardware interrupt to be triggered.**The PIC lets the processor know what **Interrupt Request (IRQ)** to call within out **Interrupt Descriptor Table (IDT)**.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **8259A Software Port Map** | | | | |
| Port Address | Description |  |  |  |
| 0x20 | Primary PIC Command and Status Register | | | |
| 0x21 | Primary PIC Interrupt Mask Register and Data Register | | | |
| 0xA0 | Secondary (Slave) PIC Command and Status Register | | | |
| 0xA1 | Secondary (Slave) PIC Interrupt Mask Register and Data Register | | | |

**Chapter 5. Basic Display Driver**

**5.1 Printing to screen**

Assuming that you are in protected mode and not using the BIOS to write text to screen, you will have write directly to "video" memory.

This is quite easy. The text screen video memory for *color*  monitors resides at 0xB8000, and for monochrome monitors it is at address 0xB0000 (see Detecting *color* and Monochrome Monitors for more information).

Text mode memory takes two bytes for every "character" on screen. One is the *ASCII code* byte, the other the *attribute* byte. so the text "HeLlo" would be stored as:

0x000b8000: 'H', *color* \_for\_H

0x000b8002: 'e', *color* \_for\_e

0x000b8004: 'L', *color* \_for\_L

0x000b8006: 'l', *color* \_for\_l

0x000b8008: 'o', *color* \_for\_o

The *attribute* byte carries the *foreground color* in its lowest 4 bits and the *background color* in its highest 3 bits. The interpretation of bit #7 depends on how you (or the BIOS) configured the hardware .

For instance, using 0x00 as attribute byte means black-on-black (you'll see nothing). 0x07 is light grey-on-black (DOS default), 0x1F is white-on-blue (Win9x's blue-screen-of-death), 0x2a is for green-monochrome nostalgic.

For color video cards, you have 16kb of text video memory to use. Since 80x25 mode does not use all 16kb (80 x 25 x 2, 4000 bytes per screen), you have 8 display pages to use.

When you print to any other page than 0, it will *not* appear on screen until that page is *enabled* or *copied* into the page 0 memory space.

**5.2 Printing strings**

If you have a pointer to video memory and want to write a string, here is how you might do it;

*/ note this example will always write to the top*

*// line of the screen*

void write\_string( int colour, const char \*string )

{

volatile char \*video = (volatile char\*)0xB8000;

while( \*string != 0 )

{

\*video++ = \*string++;

\*video++ = colour;

}

}

This simply cycles through each character in the string, and copies it to the appropriate place in video memory.

For a more advanced print function, you need to store variables for x and y, as the display controller will not print a newline. This involves a switch statement or similar construct. You also have to test for x>80 or y>25 and in the case of x>80 setting x to 0 and incrementing y, or in the case of y>25 scrolling.

### 5.3 Nothing is Displayed

Keep in mind that this way of writing to video memory will \_only\_ work if the screen has been correctly set up for 80x25 video mode (which is mode 03). You can do this either by initializing every VGA register manually, or by calling the *Set Video Mode* service of the BIOS Int10h while you're still in real mode (in your bootsector, for instance). Most BIOS's do that initialization for you, but some other (mainly on laptops) do not. Check out Ralf Brown's Interrupt List for details. Note also that some modes that are reported as "both text & graphic" by mode lists are actually graphic modes with BIOS functions that plot fonts when you call char/message output through Int10h (which means you'll end up with plain graphic mode once in Protected Mode).

(GRUB does this setup for you.)

Another common mistake, e.g. in numerous tutorials spread across the net, is to link the .text section of your kernel/OS to the wrong memory address. If you don't have memory management in place yet, make sure you're using physical memory locations in the linker script

### 5.4 Printing a Character

While in Protected Mode, try a simple command like:

// C

\*((int\*)0xb8000)=0x07690748;

// NASM

mov [0xb8000], 0x07690748

// GAS

movl $0x07690748,0xb8000

which should display 'Hi' in grey-on-black on top of your screen. If this does not work, check your paging / segmentation setup correctly maps your assumed video memory address to 0xB8000 (or 0xB0000).

**5.5 Text UI**

## 5..5.1 Popularity and Use

With the rise of graphical UI's, text based user interfaces still remain practical in hobbyist operating system projects, as they are the most easy type of interface to implement, and are still in use in larger and commercial operating systems where a graphical user interface is unnecessary, for example, in large servers. A text user interface is also easy for quickly outputting results or debug data, and inputing commands to test new features, rather than writing complicated graphical applications.

## 5.5.2 Input

Input in a text user interface primarily involves the use of a command line shell, usually through entering commands via a keyboard. Other methods do exist, such as text-based menus that can scroll up and down, and prompts asking a user to press a specific key for a specific event to occur.

## 5.5.3 Video Mode

The most used VGA video mode for a text UI is "VGA mode 3". This is the most commonly used, as it allows direct memory access to a linear address containing each character and it's associated attributes. VGA mode 3 provides a text interface 80 characters wide and 25 characters lines per screen.

## 5.5.4 Video Memory

In VGA mode 3, the linear text buffer is located in physical at 0xB8000. Reading and writing to and from this address will provide direct manipulation of on screen text. To access a particular character on the screen from X and Y coordinates is simple using the following formula:

position = (y\_position \* characters\_per\_line) + x\_position;

Each character takes up two bytes of space in memory. The first byte is split into two segments, the forecolour, and the backcolour. The second byte is a n 8-bit ASCII value of the character to print.

## 5.5.5 Colours

Each character has a colour byte. This colour byte is split up in forecolour and backcolour.

The layout of the byte, using the standard colour palette:

Bit 76543210

||||||||

|||||^^^-fore colour

||||^----fore colour bright bit

|^^^-----back colour

^--------back colour bright bit OR enables blinking Text

Its easy to write to BL, the Colour Nibbles(4Bit), in a Hex Value.   
For Example:

0x01 sets the background to black and the fore colour to blue

0x10 sets the background to blue and the fore colour to black

0x11 sets both to blue.

The default display colours set by the BIOS upon booting are 0x0F: 0 (black) for the background and 7 (White) + 8 (Bright) for the foreground.

In text mode 0, the following standard colour palette is available for use. You can change this palette with VGA commands.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Number** | **Colour** | **Name** | **Number + bright bit** | **bright Colour** | **Name** |
| 0 |  | Black | 0+8=8 |  | Dark Gray |
| 1 |  | Blue | 1+8=9 |  | Light Blue |
| 2 |  | Green | 2+8=A |  | Light Green |
| 3 |  | Cyan | 3+8=B |  | Light Cyan |
| 4 |  | Red | 4+8=C |  | Light Red |
| 5 |  | Magenta | 5+8=D |  | Light Magenta |
| 6 |  | Brown | 6+8=E |  | Yellow |
| 7 |  | Light Gray | 7+8=F |  | White |

## 5.5.6 C Code for Print a Character

The following C code will print a character at X and Y coordinates:

void WriteCharacter(unsigned char c, unsigned char forecolour, unsigned char backcolour, int x, int y)

{

uint16\_t attrib = (backcolour << 4) | (forecolour & 0x0F);

volatile uint16\_t \* where;

where = (volatile uint16\_t \*)0xB8000 + (y \* 80 + x) ;

\*where = c | (attrib << 8);

}

## 5.5.7 Scrolling

Scrolling is achieved through copying the second line of characters onto the first, the third onto the second, etc. Then clearing the last line of text with null characters (usually a blank space with the fore/back colours of 7 and 0, respectively). In text mode 0, this is accomplished by copying characters 80-159 in to memory position 0-79, 160-239 into 80-159, etc.

**Chapter 6. Keyboard**

**6.1 Keyboard history**

## 6.1.1 Back in time

A keyboard is an input device that we use as a form of input to a computer. They were originally modeled off of a typical typewriter when they were first introduced. However the creation of the keyboard was not directly modeled from it.

When the typewriter was patented in by Christopher Latham Sholes in 1877 several manufacturers and people further developed the original design. What evolved through a series of inventions was the Telegraph. Around this same time, inside of the 1930s, IBM was using Keypunches (punch card machines combined with typewriters) in their adding machines. Early computer keyboards adapted from both the keypunch and telegraph designs.

The ENIAC (Electronic Numerical Integrator And Computer) was the first general purpose computer. The ENIAC used a punchcard reader as both an input and output device. in 1946.

In 1948, the BINAC (BINary Automatic Computer) used an electromechanically controlled typewriter as both an input and output device.

When does the keyboard evolve from these inventions? The computer keyboard that we all know does not evolve into what it is today until 1964 when MIT (with Fernando Corbató), Bell Laboratories and General Electric joined together to create the Multics (Multiplexed Information and Computing Service) machine. With the Multics, a new interface was at hand: They combined the technology of the cathode ray tube (CRT) used in televisions and electric typewriters to create a Video Display Terminal (VDT). The VDTs allowed a way for the users to be able to see what they were typing which made the computer alot more easier to work with. Over the course of the 1970s and 1980s almost all computers had a form of VDT technology and a form of an electronic keyboard for input. Through the years, CRT and LCD displays replaced VDT technology, and the electronic keyboard also became standard among all general purpose computers.

Today, we use keyboards every time we go on a computer. Most of the keyboards layout still remains from the typewriter and the way it is used are the same. However, thanks to the new era of electronic devices keyboards now come in alot of different forms. From the generic plastic keyboards, keyboards the fold or have back lights in them, to even laser keyboards.

## 6.1.2 Keyboard Layout

The generic keyboard layout is known as a **QWERTY keyboard** because the characters QWERTY are the first five characters on a typical keyboard. The QWERTY layout was purposely designed during the typewriter era to slow down the typing speed of typists because of the original mechanical limitations of early typewriters. This was primarily to decrease the amount of time between each keypress and to give the print heads enough time so they do not jam.

The QWERTY layout has been adapted in all keyboards to this day.

**6.2 Inside of keyboard**

What actually happens when you press a key on your keyboard? How can the keyboard tell the program what keys are down? The very text that is being read right now (thats right, me ;) has been input by keyboard. How can the keyboard do this? *Lets take a look!*

## 6.2.1 Opening the case

You might be surprised by how keyboards came from being complex printed circuit boards (PCBs) to a single integraded board with its own microprocessor. If you were to open your keyboard, you might see something like this:

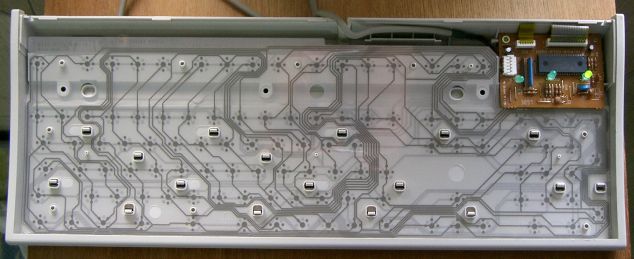


fig. Inside the case

Yep, thats it. Notice how simple this is. One circuit board and a grid. The grid might be a little hard to see in the above picture. However if you look close, you might be able to see the points in the grid and **notice that the points match to the key positions on a typical keyboard**. This is known as the **key matrix**. In almost all keyboards, the circuits that make up the key matrix is broken between each point in the grid. Knowing that a key is above a point in the key matrix, when we press down the key, it presses the switch at that point completing the horizontal circuit and allowing current to run through it. The vibration of the line caused by the mechanical movement of the keys is known as **bounce** and is filtered out by the keyboards own **microprocessor** otherwise known as the **Keyboard Encoder**. Don't worry if this seems a little complex. We will look at everything more closely in the next couple of sections.

### 6.3 Keyboard Encoder

The microprocessor used by the keyboard is useually a form of the original **Intel 8048**, which just so happens to be also Intels first microcontroller. This controller is known as the **Keyboard Encoder**. The exact keyboard encoder used is very dependent on your keyboard. There are hundereds of different keyboard encoders but they all do basically the same thing.

The rows and columns within the key grid are connected to 8 bit I/O ports on the keyboard encoder. When a key is down, the switch at that location within the key grid is closed which allows current to flow through it completing the circuit. This current enables the pin on the keyboard encoder of the correct ports that the key location corrosponds to. Thus, all the controller needs to do is scan its ports to see if a key is down or not by checking if a port line is active.

If a key is down, the keyboard encoder looks up the location within its **Read Only Memory (ROM)** character map to see what the **Scan Code** is for that character and stores it in its internal 16 byte memory. The keyboards processor includes its own timer, 33 instruction set, and can even access 128K of external memory. Using its timer, it can determin if the key is down based on weather it is by user input or a **bounce**. If a **bounce** happens, it will useually be much faster then any human can input. If the key is still down when its timer reaches 0, it is reset and the character is inserted into its internal 16 byte buffer.

**It is important to note that there are two keyboard controllers that we can communicate with: The keyboard encoder inside of the keyboard and the keyboard controller inside on the motherboard.**

### 6.4 Keyboard Controller

The keyboard controller used inside of the system case is useually a form of the original **8042 keyboard controller**. The keyboard controller interfaces with the keyboard encoder through the keyboards protocol and provides a way to interface to it. On most newer systems, the keyboard controller is not a separate **integrated circuit (IC)** but rather part of the motherboards **Super Input/Output (IO) controller** that also houses the **floppy disk controller (FDC)**, **parallel port interface**, **serial port interfaces** and **mouse interface**. Most newer systems super IO controller uses the **Low Pin Count (LPC)** bus rather then **Industry Standard Architecture (ISA)** on the southbridge of the motherboard.

### 6.5 Scan Codes

A **Scan Code** is a data packet that represents the state of a key. If a key is pressed, released, or held down, a **scan code** is sent to the computers onboard **keyboard controller**. There are two types of scan codes: **Make Codes** and **Break Codes**. A **Make Code** is sent when a key is pressed or held down while a **break code** is sent when a key is released. There is a unique make code and break code for each key on the keyboard. The entire set of numbers that represent all of the scan codes make up the keyboards **scan code set**.

There are generally three different scan sets that the keyboard can use. However there is no easy way to determin what scan set it uses as the scan values are random. Because of this, you will need to use a lookup table to determin the key the scan code represents.

## 6.6 Port Mapping

In the i86 architecture, the following ports are used to communicate with the keyboard:

|  |  |  |
| --- | --- | --- |
| **Keyboard Controller Ports** | | |
| Port | Read/Write | Descripton |
| Keyboard Encoder | | |
| 0x60 | Read | Read Input Buffer |
| 0x60 | Write | Send Command |
| Onboard Keyboard Controller | | |
| 0x64 | Read | Status Register |
| 0x64 | Write | Send Command |

This table is not to bad I hope ;) Basically: **To send a command to the keyboard encoder, write the command byte to port 0x60.** Before doing this however, you need to insure that bit 0 (output buffer full) of the keyboard controller status register is 0 to insure it is safe. If bit 1 (input buffer full) of the keyboard controller status register is 1 then data is in the input buffer ready to be read.**Reading from port 0x60 will allow you to get this data from the keyboard encoder.** The data read from the keyboard encoder will normally come from the keyboard. however you can also reprogram the microcontroller to return specific values as well.

**Writing a value to port 0x64 will allow you to send a command byte to the onboard keyboard controller. Reading from port 0x64 will allow you to get the status byte of the keyboard controller.**

*Simple Keyboard code*

*/\**

*PS/2 keyboard code.*

*Dependencies:*

*inb function and scancode table.*

*\*/*

char getScancode()

{

char c=0;

do {

if(inb(0x60)!=c)

{

c=inb(0x60);

if(c>0)

return c;

}

}while(1);

}

char getchar()

{

return scancode[getScancode()+1];

}

### 6.7 Scan Code Set 1

The following table shows which scan codes correspond to which keys when using scan code set 1 (for a "US QWERTY" keyboard only):

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Scan code** | **Key** | **Scan code** | **Key** | **Scan code** | **Key** | **Scan code** | **Key** |
|  |  | 0x01 | escape pressed | 0x02 | 1 pressed | 0x03 | 2 pressed |
| 0x04 | 3 pressed | 0x05 | 4 pressed | 0x06 | 5 pressed | 0x07 | 6 pressed |
| 0x08 | 7 pressed | 0x09 | 8 pressed | 0x0A | 9 pressed | 0x0B | 0 (zero) pressed |
| 0x0C | - pressed | 0x0D | = pressed | 0x0E | backspace pressed | 0x0F | tab pressed |
| 0x10 | Q pressed | 0x11 | W pressed | 0x12 | E pressed | 0x13 | R pressed |
| 0x14 | T pressed | 0x15 | Y pressed | 0x16 | U pressed | 0x17 | I pressed |
| 0x18 | O pressed | 0x19 | P pressed | 0x1A | [ pressed | 0x1B | ] pressed |
| 0x1C | enter pressed | 0x1D | left control pressed | 0x1E | A pressed | 0x1F | S pressed |
| 0x20 | D pressed | 0x21 | F pressed | 0x22 | G pressed | 0x23 | H pressed |
| 0x24 | J pressed | 0x25 | K pressed | 0x26 | L pressed | 0x27 | ; pressed |
| 0x28 | ' (single quote) pressed | 0x29 | ` (back tick) pressed | 0x2A | left shift pressed | 0x2B | \ pressed |
| 0x2C | Z pressed | 0x2D | X pressed | 0x2E | C pressed | 0x2F | V pressed |
| 0x30 | B pressed | 0x31 | N pressed | 0x32 | M pressed | 0x33 | , pressed |
| 0x34 | . pressed | 0x35 | / pressed | 0x36 | right shift pressed | 0x37 | (keypad) \* pressed |
| 0x38 | left alt pressed | 0x39 | space pressed | 0x3A | CapsLock pressed | 0x3B | F1 pressed |
| 0x3C | F2 pressed | 0x3D | F3 pressed | 0x3E | F4 pressed | 0x3F | F5 pressed |
| 0x40 | F6 pressed | 0x41 | F7 pressed | 0x42 | F8 pressed | 0x43 | F9 pressed |
| 0x44 | F10 pressed | 0x45 | NumberLock pressed | 0x46 | ScrollLock pressed | 0x47 | (keypad) 7 pressed |
| 0x48 | (keypad) 8 pressed | 0x49 | (keypad) 9 pressed | 0x4A | (keypad) - pressed | 0x4B | (keypad) 4 pressed |
| 0x4C | (keypad) 5 pressed | 0x4D | (keypad) 6 pressed | 0x4E | (keypad) + pressed | 0x4F | (keypad) 1 pressed |
| 0x50 | (keypad) 2 pressed | 0x51 | (keypad) 3 pressed | 0x52 | (keypad) 0 pressed | 0x53 | (keypad) . pressed |
|  |  |  |  |  |  | 0x57 | F11 pressed |
| 0x58 | F12 pressed |  |  |  |  |  |  |
|  |  | 0x81 | escape released | 0x82 | 1 released | 0x83 | 2 released |
| 0x84 | 3 released | 0x85 | 4 released | 0x86 | 5 released | 0x87 | 6 released |
| 0x88 | 7 released | 0x89 | 8 released | 0x8A | 9 released | 0x8B | 0 (zero) released |
| 0x8C | - released | 0x8D | = released | 0x8E | backspace released | 0x8F | tab released |
| 0x90 | Q released | 0x91 | W released | 0x92 | E released | 0x93 | R released |
| 0x94 | T released | 0x95 | Y released | 0x96 | U released | 0x97 | I released |
| 0x98 | O released | 0x99 | P released | 0x9A | [ released | 0x9B | ] released |
| 0x9C | enter released | 0x9D | left control released | 0x9E | A released | 0x9F | S released |
| 0xA0 | D released | 0xA1 | F released | 0xA2 | G released | 0xA3 | H released |
| 0xA4 | J released | 0xA5 | K released | 0xA6 | L released | 0xA7 | ; released |
| 0xA8 | ' (single quote) released | 0xA9 | ` (back tick) released | 0xAA | left shift released | 0xAB | \ released |
| 0xAC | Z released | 0xAD | X released | 0xAE | C released | 0xAF | V released |
| 0xB0 | B released | 0xB1 | N released | 0xB2 | M released | 0xB3 | , released |
| 0xB4 | . released | 0xB5 | / released | 0xB6 | right shift released | 0xB7 | (keypad) \* released |
| 0xB8 | left alt released | 0xB9 | space released | 0xBA | CapsLock released | 0xBB | F1 released |
| 0xBC | F2 released | 0xBD | F3 released | 0xBE | F4 released | 0xBF | F5 released |
| 0xC0 | F6 released | 0xC1 | F7 released | 0xC2 | F8 released | 0xC3 | F9 released |
| 0xC4 | F10 released | 0xC5 | NumberLock released | 0xC6 | ScrollLock released | 0xC7 | (keypad) 7 released |
| 0xC8 | (keypad) 8 released | 0xC9 | (keypad) 9 released | 0xCA | (keypad) - released | 0xCB | (keypad) 4 released |
| 0xCC | (keypad) 5 released | 0xCD | (keypad) 6 released | 0xCE | (keypad) + released | 0xCF | (keypad) 1 released |
| 0xD0 | (keypad) 2 released | 0xD1 | (keypad) 3 released | 0xD2 | (keypad) 0 released | 0xD3 | (keypad) . released |
|  |  |  |  |  |  | 0xD7 | F11 released |
| 0xD8 | F12 released |  |  |  |  |  |  |
| 0xE0, 0x1C | (keypad) enter pressed | 0xE0, 0x1D | right control pressed |  |  |  |  |
|  |  | 0xE0, 0x35 | (keypad) / pressed |  |  |  |  |
| 0xE0, 0x38 | right alt (or altGr) pressed |  |  |  |  |  |  |
|  |  |  |  |  |  | 0xE0, 0x47 | home pressed |
| 0xE0, 0x48 | cursor up pressed | 0xE0, 0x49 | page up pressed |  |  | 0xE0, 0x4B | cursor left pressed |
|  |  | 0xE0, 0x4D | cursor right pressed |  |  | 0xE0, 0x4F | end pressed |
| 0xE0, 0x50 | cursor down pressed | 0xE0, 0x51 | page down pressed | 0xE0, 0x52 | insert pressed | 0xE0, 0x53 | delete pressed |
|  |  |  |  |  |  | 0xE0, 0x5B | left GUI pressed |
| 0xE0, 0x5C | right GUI pressed | 0xE0, 0x5D | "apps" pressed |  |  |  |  |
| 0xE0, 0x9C | (keypad) enter released | 0xE0, 0x9D | right control released |  |  |  |  |
|  |  | 0xE0, 0xB5 | (keypad) / released |  |  |  |  |
| 0xE0, 0xB8 | right alt (or altGr) released |  |  |  |  |  |  |
|  |  |  |  |  |  | 0xE0, 0xC7 | home released |
| 0xE0, 0xC8 | cursor up released | 0xE0, 0xC9 | page up released |  |  | 0xE0, 0xCB | cursor left released |
|  |  | 0xE0, 0xCD | cursor right released |  |  | 0xE0, 0xCF | end released |
| 0xE0, 0xD0 | cursor down released | 0xE0, 0xD1 | page down released | 0xE0, 0xD2 | insert released | 0xE0, 0xD3 | delete released |
|  |  |  |  |  |  | 0xE0, 0xDB | left GUI released |
| 0xE0, 0xDC | right GUI released | 0xE0, 0xDD | "apps" released |  |  |  |  |
|  |  |  |  |  |  | 0xE0, 0x2A, 0xE0, 0x37 | print screen pressed |
|  |  |  |  |  |  | 0xE0, 0xB7, 0xE0, 0xAA | print screen released |
|  |  | 0xE1, 0x1D, 0x45, 0xE1, 0x9D, 0xC5 | pause pressed |  |  |  |  |

There is no scan code for "pause key released" (it behaves as if it is released as soon as it's pressed)