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.

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.

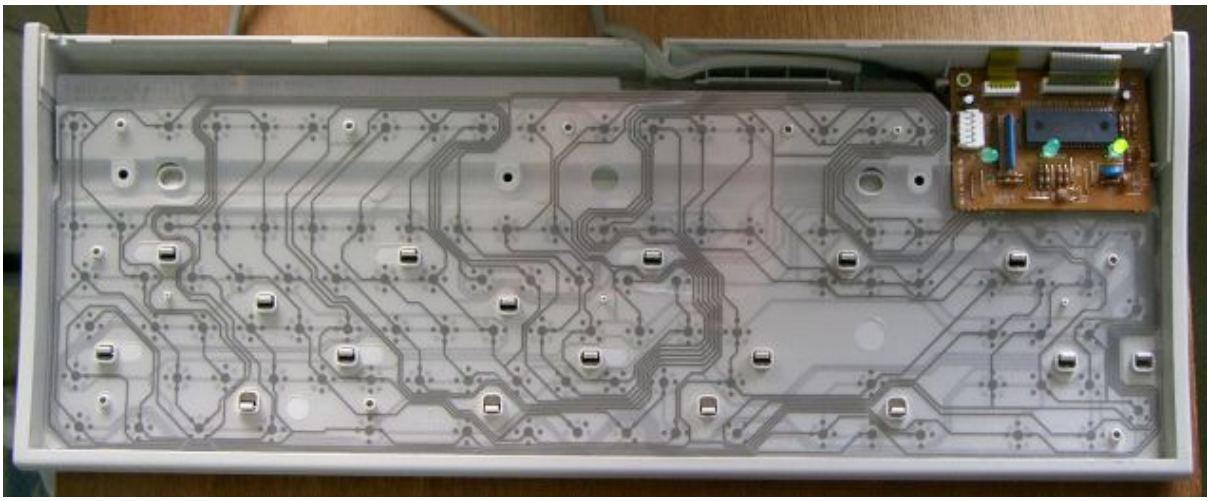
Inside the keyboard

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!*

Note: The exact details depend on the keyboards specific type and model. Because of this, I will only be covering a generic 102 key keyboard here.

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:



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.

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.

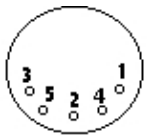
It is important to note that there are two keyboard controllers that we can communicate with: The keyboard encoder inside of the keyboard and the keyboard controller inside on the motherboard. We will look at the other controller a little later, don't worry ;) For now, keep in mind that their are two controllers, **and the keyboard encoder is one of them**.

The keyboard encoder communicates with the system through a method defined by the **keyboard protocol**. Lets take a look.

Keyboard Protocol

The keyboard encoder sends data as bytes to the motherboards onboard keyboard controller. The way it is sent depends on the **protocal** used by the keyboards interface. This is useually a 5-pin DIN connector, 6 pin Mini-DIN connector, USB connector, SDL connector, or wireless using an infired (IR) interface.

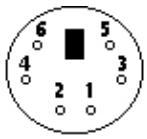
The 5 pin DIN connector used for AT/XT keyboards normally is on the back of the computer and looks like this:



1: Clock 2: Data 3: N/A 4: Ground 5: Vcc (+5V)

The motherboard supplies power from the power supply unit (PSU) through the Vcc and Ground pins. The clock pin is used for synchronization between the keyboards data and the system clock. The data from the keyboard is sent as serial data over the data pin.

The more common 6 pin Mini-DIN connector used for PS/2 keyboards is very simular:



1: Data 2: N/A 3: Ground 4: Vcc (+5V) 5: Clock 6: N/A

Nothing much new to add here. DIN does not really stand for anything in particular but refers to the standards group that developed it (Deutsches Institut für Normung, or in English, German Institute for Standardization).

The SDL (Shielded Data Link) connector is very simular:

A: N/A B: Data C: Ground D: Clock E: Vcc (+5V) F: N/A

Universal Serial Bus (USB) connectors are a standard that is used by alot of different devices. Working with USB devices directly are a fairly complex topic. They only contain four pins: 1: Vcc (+5V), 2: Data-, 3: Data+, 4: Ground.

USB Legacy Support is used on most modern computers that come with USB ports. This means that these computer motherboards can emulate USB keyboards and mice as PS/2 keyboard and mice. Because of this: **Communicating with a USB keyboard or mice using PS/2 compatable interfaces will work.** In other words, do not worry if you have a USB keyboard or mouse as most of us do. The code and demo in this tutorial will still work fine thanks to the emulation provided by the motherboard.

As you can see, the interfaces between the keyboard and the computer are not too complex. All they do is provide a way to send data as bits between the keyboard controller and the keyboard encoder. The data are

routed to the onboard or integrated keyboard controller on the motherboard. The keyboard controller takes control.

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.

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.

Lets take a look at the scan code tables. **Note: These tables are important! We will need to use these to determin what keys are pressed on the keyboard.** Also note: All scan codes in these tables in are hexidecimal.

These are fairly large tables so I decided to put them as a separate resource. Please see the tables in the resources section here.

Lets have an example. If you press shift+A keys on your keyboard, what will be the make code sent to your computer? In order to better understand this, lets take a look at the sequence of events that happens. First the shift key is pressed, then the A key is pressed. Then the A key is released followed by the shift key being released. Assuming that the scan code set is the default scan code set for modern keyboards, the left shift key make code is 0x12, break code is 0xF0 and 0x12. The make code for the A key is 0x1C while the break code is 0xF0 and 0x1C. So when this event occurs, the following scan codes will be sent to the computer:

```
Key events: shift down  A down  A released  Shift released
Scan codes: 0x12        0x1C    0xF0 0x1C   0xF0 0x12
```

Looking at the above, we can see that the scan codes sent will be 0x12, 0x1C, 0xF0, 0x1C, 0xF0, and 0x12.

If you press a key and hold it, the key becomes **typematic**. In other words, the keyboard will keep sending the keys make code until the key is released or another key is pressed. Try it: Open up your favoriate text editor and hold down a key. After a short delay another of the same character will appear followed by a long series of the character. The **typematic delay** determins the amount of seconds to wait before entering typematic mode, and the **typematic rate** determins the amount of character make codes per second to send to the computer. During typematic mode, the character data is not buffered. If multiple keys are held down, only the last key held becomes typematic.

Scan codes are very important to us. When a scan code is sent to the onboard keyboard controller, the keyboard controller stores the scan code into its internal memory. The keyboard controller then toggles its **Interrupt Request (IR)** line to high. If the interrupt line is not masked by the **Programmable Interrupt Controller (PIC)** then this will cause **IRQ 1** to be fired. Even if the IRQ is masked, because the read buffer can be read by us through software, we can read the scan code and determin what key was just released or pressed.

Keyboard Interface: Developing a Device Driver

We have covered alot already in this chapter. We have looked at the history of the keyboard as in interface device, the QWERTY keyboard layout, and looked at the inside of the keyboard to see how it works and the primary components that make it work. We have also looked at scan code sets and the keyboards protocols. Don't worry if you do not understand everything yet, we will look at everything in more detail within the next couple of sections. We will also be developing device driver for our keyboard as well. Cool, huh? All of the code in this section will also be in the final demo.

Keyboard Interfacing: Polling

Remember from the previous section that there are **two** controllers when working with the keyboard? That is, there is the **Keyboard Encoder** inside of the keyboard as well as the **Keyboard Controller** on the motherboard. This is the first chapter in the series where we need to interface with several different controllers to control a single hardware device. Thats right: We can communicate with **both** of these controllers. Well, kind of. When we send a command to the keyboard encoder, we still send it to the onboard keyboard controller however it reroutes it to the keyboard encoder over the keyboard protocol.

Okay, so we can communicate with both controllers. How fun is that? Knowing that both controllers work with each other, they also communicate with each other. The keyboard encoder may send alot of different codes to the onboard keyboard controller to store. These can be scan codes or error codes. This allows us to also recieve information from both the keyboard encoder and onboard controller.

All of this communication is done by simply using the IN and OUT instructions to read or write to the controller ports mapped in the IO address space. While we never had to worry what these ports are, understanding how IO mapping works with controllers becomes more important here.

This is one way we can interface with the keyboard: We can manually communicate with the controllers to check if a key is down, up, or what not. This is known as **polling** the keyboard. This is how we are able to get the last scan code from the keyboard: by polling the keyboard controller for it.

Keyboard Interfacing: Interrupt Request (IRQ)

Remember from the PIC tutorial that the keyboard controller can be configured to use an interrupt line? We can configure the keyboard controller to issue IRQ 1 whenever a key has been pressed or released. This is the most common way to interface with the keyboard.

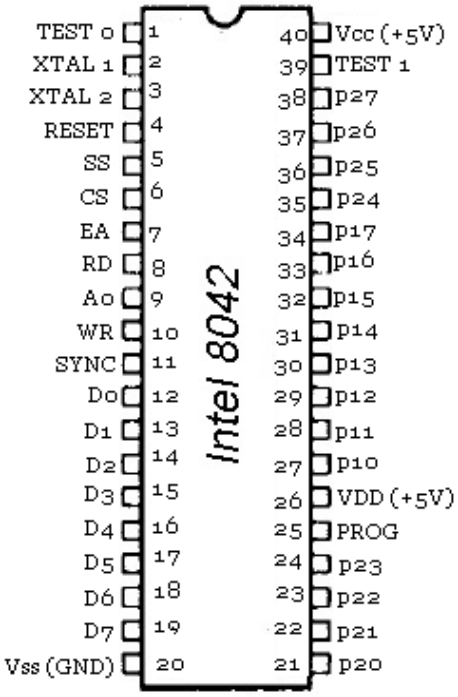
Whenever IRQ 1 is fired, you should always test to see if a scan code has actually been sent to the keyboard controller. This is done by polling the keyboard controller to get the last scan code.

Detail: 8042 Keyboard Microcontroller

The original 8042 Microcontroller

This is the microcontroller that interfaces with the keyboard encoder. The keyboard controller is part of a family of microcontrollers originally started with the 8042 microcontroller. In modern computers, the keyboard controller is not a separate **integrated circuit (IC)** but its functionality is emulated by the motherboard itself. That is, the controller functionalty is integrated into the motherboard chipset.

The keyboard controller can operate in two modes: **AT Compatable Mode** and **PS/2 Compatable Mode**. Depending on what mode the controller is operating in, the way it interfaces to the outside world may differ. First, lets take a look at the controller:



Yep. Thats it. Pins **P10-P17** is the controllers **input port**. Pins **P20-P27** is the controllers **output port**. And pin **T0-T1** is the controller **test port**. The exact meaning of these pins depend on the mode of operation the controller is in.

We will look closer at these ports later as there are commands that allow us to work with them.

Most of the other pins are not important to us. I decided to add them here for completeness sake only; you do not need to know them.

The pins **XTAL 1** and **XTAL 2** are for **Crystal Oscillator Input**. XTAL 1 can also be connected to ground if CLK is driven by an external source. Simularily, **XTAL 2** can be connected to **CLK** if its being driven by an external source.

RESET causes the controller to reset if pulled low (0).

SS is the microcontrollers **Single Step** pin. **CS** is the **Chip Select** pin which is used for data register port interfacing. **EA** (No, not the company ;)) is the **External Access** input pin. This will disable the **OTP (One Time Programmable) ROM** and enable commands to be sent to the controller from an external source.

RD Output enable input; used for data register port interfacing. **A0** is the Command/Data Register select input; used for data register port interfacing. **WR** is the write enable input line, used with data register port interfacing.

SYNC is the clock output signal. **D0-D7** is used for data register port interfacing. **GND** is the ground pin (Vss). **Vdd** is the +5V input pin. **PROG** is used as an address/data strobe to the 8243 during I/O expander access. **Vcc** is another +5V input pin.

The keyboard controller provides us an interface for controlling how we want the keyboard to work. We do this by communicating with the keyboard controller through its ports which is mapped into the port I/O space. As you know, this means in order for us to communciate with the keyboard controller, we will need to use the IN and OUT instructions and know how it is mapped. Lets take a look.

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.

Knowing all of this, we can easily provide routines for reading and writing command bytes and data to and from these controllers. We abstract the IO ports used by these controllers here:

```
enum KYBRD_ENCODER_IO {

        KYBRD_ENC_INPUT_BUF     =       0x60,
        KYBRD_ENC_CMD_REG       =       0x60
};

enum KYBRD_CTRL_IO {

        KYBRD_CTRL_STATS_REG    =       0x64,
        KYBRD_CTRL_CMD_REG      =       0x64
};
```

We will not be going over the routines used to interact with these controllers just yet as it requires some knowledge of the commands of the controllers.

Registers

Status Register

This might look familiar from When we covered enabling the 20th address line. To read the status register, simply read from I/O port 0x64. The value returned is an 8 bit value that follows a specific format. The format is a little different depending on the mode of the controller.

Here it is again. I **bolded** the important ones:

- **Bit 0: Output Buffer Status**
  - 0: Output buffer empty, dont read yet
  - 1: Output buffer full, please read me :)
- **Bit 1: Input Buffer Status**
  - 0: Input buffer empty, can be written
  - 1: Input buffer full, dont write yet
- **Bit 2:** System flag
  - 0: Set after power on reset
  - 1: Set after successfull completion of the keyboard controllers self-test (Basic Assurance Test, BAT)
- **Bit 3:** Command Data
  - 0: Last write to input buffer was data (via port 0x60)
  - 1: Last write to input buffer was a command (via port 0x64)
- **Bit 4:** Keyboard Locked
  - 0: Locked
  - 1: Not locked
- **Bit 5:** Auxiliary Output buffer full
  - PS/2 Systems:
    - 0: Determins if read from port 0x60 is valid If valid, 0=Keyboard data
    - 1: Mouse data, only if you can read from port 0x60
  - AT Systems:
    - 0: OK flag
    - 1: Timeout on transmission from keyboard controller to keyboard. **This may indicate no keyboard is present.**
- **Bit 6:** Timeout
  - 0: OK flag
  - 1: Timeout
  - PS/2:
    - General Timeout
  - AT:

    - Timeout on transmission from keyboard to keyboard controller. **Possibly parity error (In which case both bits 6 and 7 are set)**
- **Bit 7:** Parity error
    - 0: OK flag, no error
    - 1: Parity error with last byte

We need to read the status register to determin the current state of the keyboard and to see what we can and cannot do. For example, we would not want to send a command to the keyboard if there is no keyboard plugged in! So we would want to read in the current status to test it before sending a command.

We also need to take into coinsideration that the processor executes instructions alot faster then the keyboard controller can respond. Because of this, there are alot of times when we need to wait for the keyboard controller to be ready for the next command. To check this, we need to read in the status register and test bit 0 (Output Buffer Full) to see if it is okay to send the next command. If we do not do this, the previous command will be discarded and the new one will start executing, which may not be desirable.

It is important to wait for the controller to be ready before sending another command or reading data from it.

We can use bit masks for reading and writing to the status register. Here is the one from the demo at the end of this chapter. Notice how each bit coorosponds with the correct bit in the list shown above.

```
enum KYBRD_CTRL_STATS_MASK {

        KYBRD_CTRL_STATS_MASK_OUT_BUF    =       1,              //00000001
        KYBRD_CTRL_STATS_MASK_IN_BUF     =       2,              //00000010
        KYBRD_CTRL_STATS_MASK_SYSTEM     =       4,              //00000100
        KYBRD_CTRL_STATS_MASK_CMD_DATA   =       8,              //00001000
        KYBRD_CTRL_STATS_MASK_LOCKED     =       0x10,           //00010000
        KYBRD_CTRL_STATS_MASK_AUX_BUF    =       0x20,           //00100000
        KYBRD_CTRL_STATS_MASK_TIMEOUT    =       0x40,           //01000000
        KYBRD_CTRL_STATS_MASK_PARITY     =       0x80            //10000000
};
```

Great! So, all we need to do is to read from the keyboard controllers status register at port 0x64. Then test whatever bit we want to check the status of it based on the bit masks above.

So, to read from the keyboard controller status register, all we need is:

```
//! read status from keyboard controller
uint8_t kybrd_ctrl_read_status () {

        return inportb (KYBRD_CTRL_STATS_REG);
}
```

Reading and writing: Input buffer

To send a command, we first wait to insure the keyboard controller is ready for it. This is done by seeing if the input buffer is full or not. We test this by reading the keyboard controllers status register and testing the bit. If its 0, the buffer is empty so we send the command byte to it. (Remember all of this information is inside of the status register bit layout shown above.)

```
//! send command byte to keyboard controller
void kybrd_ctrl_send_cmd (uint8_t cmd) {

        //! wait for kkybrd controller input buffer to be clear
        while (1)
                if ( (kybrd_ctrl_read_status () & KYBRD_CTRL_STATS_MASK_IN_BUF) == 0)
                        break;

        outportb (KYBRD_CTRL_CMD_REG, cmd);
}
```

The keyboard encoder is very simular as you can see below. **Remember that commands sent to the keyboard encoder are sent to the keyboard controller first.** Because of this, you still need to insure the keyboard controller itself is still ready for the command.

```
//! read keyboard encoder buffer
uint8_t kybrd_enc_read_buf () {

        return inportb (KYBRD_ENC_INPUT_BUF);
}

//! send command byte to keyboard encoder
void kybrd_enc_send_cmd (uint8_t cmd) {

        //! wait for kkybrd controller input buffer to be clear
        while (1)
                if ( (kybrd_ctrl_read_status () & KYBRD_CTRL_STATS_MASK_IN_BUF) == 0)
                        break;

        //! send command byte to kybrd encoder
        outportb (KYBRD_ENC_CMD_REG, cmd);
}
```

Keyboard Encoder Commands

When writing a command byte to port 0x60 the keyboard controller transmits the value directly to the keyboard encoder. The following is a list of the command bytes:

Command Listing	
Command	Descripton
0xED	Set LEDs
0xEE	Echo command. Returns 0xEE to port 0x60 as a diagnostic test
0xF0	Set alternate scan code set
0xF2	Send 2 byte keyboard ID code as the next two bytes to be read from port 0x60
0xF3	Set autrepeat delay and repeat rate
0xF4	Enable keyboard
0xF5	Reset to power on condition and wait for enable command
0xF6	Reset to power on condition and begin scanning keyboard
0xF7	Set all keys to autorepeat (PS/2 only)
0xF8	Set all keys to send make code and break code (PS/2 only)
0xF9	Set all keys to generate only make codes
0xFA	Set all keys to autorepeat and generate make/break codes
0xFB	Set a single key to autorepeat
0xFC	Set a single key to generate make and break codes
0xFD	Set a single key to generate only break codes
0xFE	Resend last result
0xFF	Reset keyboard to power on state and start self test

All of the small commands are desribed in the above table. Lets take a closer look at the more complex commands, shall we?

Command 0xED - Set Light Emetting Diods (LED's)

This command is used to set the LEDs on the keyboard. The next byte written to port 0x60 updates the LEDs on the keyboard and follows the format shown below:

- **Bit 0:** Scroll lock LED (0: off 1:on)
- **Bit 1:** Num lock LED (0: off 1:on)
- **Bit 2:** Caps lock LED (0: off 1:on)

All other bits must be 0.

This command is kind of fun to play with ;) Here is an example routine that the demo uses to update the lights on your keyboard. Notice how it sets or clears the bit based on if a paramamter is true or false. Also notice that it writes first the command byte to the keyboard encoder followed by the data byte. They both go to the keyboard encoders command register. KYBRD_ENC_CMD_SET_LED is a constant for 0xED -- the command byte that we are using. No magic involved :)

```
//! sets leds
void kkybrd_set_leds (bool num, bool caps, bool scroll) {

        uint8_t data = 0;

        //! set or clear the bit
        data = (scroll) ? (data | 1) : (data & 1);
        data = (num) ? (num | 2) : (num & 2);
        data = (caps) ? (num | 4) : (num & 4);

        //! send the command -- update keyboard Light Emetting Diods (LEDs)
        kybrd_enc_send_cmd (KYBRD_ENC_CMD_SET_LED);
        kybrd_enc_send_cmd (data);
}
```

Command 0xF0 - Set alternatate scan code set (PS/2 Only)

This command sets the scan code set to use. The next byte written to port 0x60 must be a byte of the following format:

- **Bit 0:** Returns current scan code set to port 0x60
- **Bit 1:** Sets scan code set 1
- **Bit 2:** Sets scan code set 2
- **Bit 3:** Sets scan code set 3

All other bits should be 0.

Command 0xF3 - Set autorepeat delay and repeat rate

This command sets the autorepeat delay and repeat rate. Next byte written to port 0x60 must be the following format:

- **Bit 0-4:** Repeat rate. 0: approx 30 chars/sec to 0x1F: approx 2 chars/sec
- **Bit 5-6:** Repeat delay. 00: 1/4 sec, 01: 1/2 sec, 10: 3/4 sec, 11: 1 sec

All other bits must be 0.

Return Codes

As you know, the keyboard encoder communicates with the systems onboard keyboard controller. Most of the values returned will be a scan code but sometimes it may also return an error. These values are sent from the keyboard decoder to the system through port 0x60.

The returned value can be one of the following:

Return Codes	
Value	Descripton
0x0	Internal buffer overrun
0x1-0x58, 0x81-0xD8	Keypress scan code
0x83AB	Keyboard ID code returned from F2 command
0xAA	Returned during **Basic Assurance Test (BAT)** after reset. Also L. shift key make code
0xEE	Returned from the ECHO command
0xF0	Prefix of certain make codes (Does not apply to PS/2)
0xFA	Keyboard acknowledge to keyboard command
0xFC	**Basic Assurance Test (BAT)** failed (PS/2 only)
0xFD	Diagonstic failure (Except PS/2)
0xFE	Keyboard requests for system to resend last command
0xFF	Key error (PS/2 only)

Onboard Keyboard Controller Commands

Some of these commands we have already seen in the A20 chapter. Alot of the commands listed here are new however and some are very low level. That is, some of these commands allow you to control specific lines

connected to the controller. This is why I had to cover the controller's lines and how it interfaces with the keyboard device. Other commands allow you to read or write to the controllers internal RAM.

Command Listing	
Command	Descripton
Common Commands	
0x20	Read command byte
0x60	Write command byte
0xAA	Self Test
0xAB	Interface Test
0xAD	Disable Keyboard
0xAE	Enable Keyboard
0xC0	Read Input Port
0xD0	Read Output Port
0xD1	Write Output Port
0xE0	Read Test Inputs
0xFE	System Reset
0xA7	Disable Mousr Port
0xA8	Enable Mouse Port
0xA9	Test Mouse Port
0xD4	Write To Mouse
Non Standard Commands	
0x00-0x1F	Read Controller RAM
0x20-0x3F	Read Controller RAM
0x40-0x5F	Write Controller RAM
0x60-0x7F	Write Controller RAM
0x90-0x93	Synaptics Multiplexer Prefix
0x90-0x9F	Write port 13-Port 10
0xA0	Read Copyright
0xA1	Read Firmware Version
0xA2	Change Speed
0xA3	Change Speed
0xA4	Check if password is installed
0xA5	Load Password
0xA6	Check Password
0xAC	Disagnostic Dump
0xAF	Read Keyboard Version
0xB0-0xB5	Reset Controller Line
0xB8-0xBD	Set Controller Line
0xC1	Continuous input port poll, low
0xC2	Continuous input port poll, high
0xC8	Unblock Controller lines P22 and P23
0xC9	Block Controller lines P22 and P23
0xCA	Read Controller Mode
0xCB	Write Controller Mode
0xD2	Write Output Buffer
0xD3	Write Mouse Output Buffer
0xDD	Disable A20 address line
0xDF	Enable A20 address line
0xF0-0xFF	Pulse output bit

Thats alot of commands, huh? It would take a very long time to cover every command here, don't you think? We have already covered the A20 commands in the ealier A20 chapter. Because portability is a concern within this series, we will only be covering the more common commands listed above. However I encourage our interested readers to look for information on the commands not covered here.

I am not going to cover example code until the next section. Rather we will just be looking at the commands themselves here and refer to them from the next section.

Command 0x20 - Read Command Byte and reading controller RAM

Look at the table above. Notice that the commands 0x20 - 0x3F is used to read the controller RAM? And yet, command 0x20 is used to read the command byte also. What is going on here?

Actually, they both refer to the same thing. The command byte is stored within the controllers RAM. So, when reading the command byte, you are reading from the controllers internal RAM. Cool?

When reading from the controllers RAM, **the last 6 bits of the command refer to the the location within RAM to read from**. On certain MCA systems, you have access to all 32 locations within the RAM. On other systems, you can only access the bytes at 0, 0x13-0x17, 0x1D, and 0x1F.

These locations are:

- **Offset 0: Command Byte**
- Offset 0x13 (MCA): nonzero when password is enabled
- Offset 0x14 (MCA): nonzero when password is matched
- Offsets 0x16-0x17 (MCA): give two make codes to be discarded during password matching
- Offset 0x1D:
- Offset 0x1F:

The **command byte** is the more important one here. It follows a specific bit format shown here. Don't worry - its not as complex as it looks:

- **Bit 0**: Keyboard interrupt enable
  - 0: Disables interrupt
  - 1: Sends IRQ 1 when keyboard output buffer is full
- **Bit 1**: Mouse interrupt enable
  - **ISA**: Unused
  - **EISA / PS2**
    - 0: Disables mouse interrupts
    - 1: Sends IRQ 12 when mouse output buffer is full
- **Bit 2**: System Flag (Also bit 2 of status register)
  - 0: Cold reboot
  - 1: Warm reboot (BAT already completed)
- **Bit 3**: Ignore Keyboard Lock
  - **PS/2**: Unused
  - **AT**
    - 0: No action
    - 1: Force bit 4 of status register to 1 (not locked)
- **Bit 4**: Keyboard Enable
  - 0: Enable Keyboard
  - 1: Disable Keyboard by driving clock line low
- **Bit 5**: Mouse Enable
  - **EISA or PS/2**
    - 0: Enable Mouse
    - 1: Disable mouse by driving clock line low
  - **ISA**
    - 0: In PC Mode, use 11-bit codes, check parity and do scan conversion
    - 1: In PC Mode, use 8086 codes, don't check parity and don't do scan conversion
- **Bit 6**: Translation
  - 0: No translation
  - 1: Translate key scancodes. MCA type 2 controllers cannot set this bit
- **Bit 7**: Unused, should be 0

I do not think we will be needing this command so I have not written a routine for it.

Command 0x60 - Write Command Byte and writing controller RAM

The command bytes 0x60 - 0x7F are very simular to the above and allows you to write to the same RAM locations as described above. the more important one is reading byte 0 of the controllers RAM (The command byte, remember?) which can be done by sending command byte 0x60.

Along with the above command, there is no routine written for this command for the end demo.

Command 0xAA - Self Test

This command causes the controller to perform a self test. It returns the result in the output buffer that can be read through port 0x60. It returns 0x55 if the test passed successfully, or 0xFC if it failed.

Here is an example routine. Notice how it first sends the KYBRD_CTRL_CMD_SELF_TEST command (command 0xAA) to the keyboard controller. Afterwords, it waits for the keyboard controllers output buffer to be filled with data. This will tell us if the test completed or not. When it completes, it returns true (test successful) if the result in the output buffer is 0x55, or false (test failed) otherwise.

```
//! run self test
bool kkybrd_self_test () {

	//! send command
	kybrd_ctrl_send_cmd (KYBRD_CTRL_CMD_SELF_TEST);

	//! wait for output buffer to be full
	while (1)
		if (kybrd_ctrl_read_status () & KYBRD_CTRL_STATS_MASK_OUT_BUF)
			break;

	//! if output buffer == 0x55, test passed
	return (kybrd_enc_read_buf () == 0x55) ? true : false;
}
```

Command 0xAB - Interface Test

This command causes the controller to test the serial interface between the controller and the keyboard. The result of the test is placed in the output buffer that can be read on port 0x60.

The result can be one of the following:

- 0: Success, no errors
- 1: Keyboard clock line stuck low
- 2: Keyboard clock line stuck high
- 3: Keyboard data line stuck high
- 0xFF: General error

As you can see, all of these are hardware errors. If an errors occurs, it is recommended to disable the keyboard and reset it. If it still fails, the keyboard might have malfunctioned.

Command 0xAD - Disable Keyboard

This command causes the controller to disable the keyboard clock line and set bit 4 (keyboard enable) of the command byte. Please see the **Read Command Byte** section for the format of the command byte.

In other words, this command disables the keyboard.

It is a good idea to store the current state of the keyboard so that your system can keep track of the current status of the keyboard. This is done in the demos keyboard driver through _kkybrd_disable.

```
//! disables the keyboard
void kkybrd_disable () {

	kybrd_ctrl_send_cmd (KYBRD_CTRL_CMD_DISABLE);
	_kkybrd_disable = true;
}
```

Command 0xAE - Enable Keyboard

This command causes the controller to enable the keyboard clock line and clears bit 4 (keyboard enable) of the command byte. Please see the **Read Command Byte** section for the format of the command byte.

In other words, this command enables the keyboard.

Here is an example routine taken from the demo. Notice how easy this one is :)

```
//! enables the keyboard
void kkybrd_enable () {

        kybrd_ctrl_send_cmd (KYBRD_CTRL_CMD_ENABLE);
        _kkybrd_disable = false;
}
```

Command 0xC0 - Read Input Port

This command reads the input port (lines **P10-P17** on the controller) and copies the binary value to the output buffer that can be read through port 0x64. Because we have not looked at the lines that this port has, lets look at it now:

- **Line P10 / Bit 0:** Keyboard data in, Unused in ISA
- **Line P11 / Bit 1:** Mouse data in, Unused in ISA
- **Line P12 / Bit 2:** Unused in ISA, EISA, PS/2
- **Line P13 / Bit 3:** Unused in ISA, EISA, PS/2
- **Line P14 / Bit 4:** 0: 512 KB motherboard RAM, 1: 256K RAM
- **Line P15 / Bit 5:** 0: Manufacturing jumper installed, 1: Not installed
- **Line P16 / Bit 6:** 0: CGA display 1: MDA display
- **Line P17 / Bit 7:** 0: Keyboard locked 1: Not locked

If a jumper is active, the BIOS may run an infinity diagnositic loop. Lines P13 and P14 may be configured for clock switching.

Don't worry if these seems complex -- Looking at the above you can probably see that this command is not very helpful on modern computers. Bits 0, 1, 2, and 3 are no longer used anymore. Bit 4 is almost useless as all modern computers have way more then 512 KB of RAM. Bit 5 can be used to test if a jumper is installed for keyboard testing (Which almost no user will do). Bit 6 is not needed as you can get that information from the video adapter and bit 7 is almost never needed as most users would not want their keybord locked. Yep. very useful command, huh? It can be, just not for most computers.

Because of the super usefulness (or lack there of ;)) this command has to offer, I decided not to write a routine for it.

Command 0xD0 - Read Output Port

This command tells the controller to read from its output port (P2) and place the result in the output buffer at port 0x64. By reading from port 0x64 after issuing this command, we can check the bits of the controllers output port.

The controllers output port is just the **P20-P27** lines of the controller (Remember this from before?). The binary value on these lines are then stored into the output buffer when this command is executed.

We have not covered the output port pins and what they are yet. (Well, actually we have in the A20 chapter, but not in detail) so lets look at it here:

- **Line P20 / Bit 0:** 0: Reset CPU, 1: normal operation
- **Line P21 / Bit 1:** 0: A20 line is forced, 1: enabled
- **Line P22 / Bit 2:** Mouse data. Unused in ISA
- **Line P23 / Bit 3:** Mouse clock. Unused in ISA
- **Line P24 / Bit 4:** 0: IRQ 1 not active, 1: IRQ 1 active
- **Line P25 / Bit 5:** 0: IRQ 12 not active, 1: IRQ 12 active
- **Line P26 / Bit 6:** Keyboard Clock
- **Line P27 / Bit 7:** Data to Keyboard

Thats it. Bit 2 and 3 are no longer used in **Industry Standard Architecture (ISA)** computers (most modern computers). Bits 4 and 5 (lines P24 and P25) are connected to the **Programmable Interrupt Controller (PIC)** on the PIC lines IR1 and IR12. Thus, if the line is active the interrupt line is also active on the PIC (Which also means the interrupt may be executing or pending execution.) Bits 6 and 7 simply contain the current keyboard clock and data signal (Whether the line is active or not.)

So far alot of the bits here are pretty useless, huh? Alot of these bits are at the electronics level of the current operation of the controller which is useless for our needs. That is, except for the first two lines (Bits 0 and 1) which control if we want to reset the system or enable/disable the 20th address line. Well, okay, reading bit 0 is useless also as the line *must* be active (1) meaning we are running in normal operation. Without it the system would reboot. Thus the only useful bit here is the A20 line. This is true for at least a read operation.

When this command is issued on port 0x64, the resulting byte is placed in the output buffer and can be read by reading the byte from port 0x60.

Im not worried about needing to disable A20 anytime soon. Also, there are alternative methods of resetting the system through the keyboard. Because of this, this command is pretty useluess for our needs and I decided not to write a routine for it.

Command 0xD1 - Write Output Port

This command copies the byte from the output buffer (port 0x60) and places the byte on the controllers output port lines. Please see the previous section (Read Output Port Command) for a description of these lines.

For the most part, you would want to bitwise OR specific bits that you would like to change and keep everything else unchanged to prevent possible problems.

This command is useful in several ways. It allows you to enable or disable the IRQ used by the controller, enable or disable the A20 gate, or even reset the system by setting bit 0. Again, please see the previous section for the list of the bits that can be changed.

Command 0xE0 - Read Test Input

This command copies takes the binary value from the test port lines on the controller and places them in the output buffer so they can be read through port 0x60.

The test port are the lines **TEST 0** and **TEST 1** of the microcontroller (please see the controller pinout diagram in this chapter). Because we have not covered them yet, lets take a look:

- **Line TEST 0 / Bit 0:** Keyboard Clock (input)
- **Line TEST 1 / Bit 1:** AT - Keyboard data (input) PS/2 - Mouse clock (input)

All other bits should be assumed undefined and should not be read.

While this command might not be much of use; remember that the controller may be used in other fields where the test port may be more useful. After all, it is there for testing purposes.

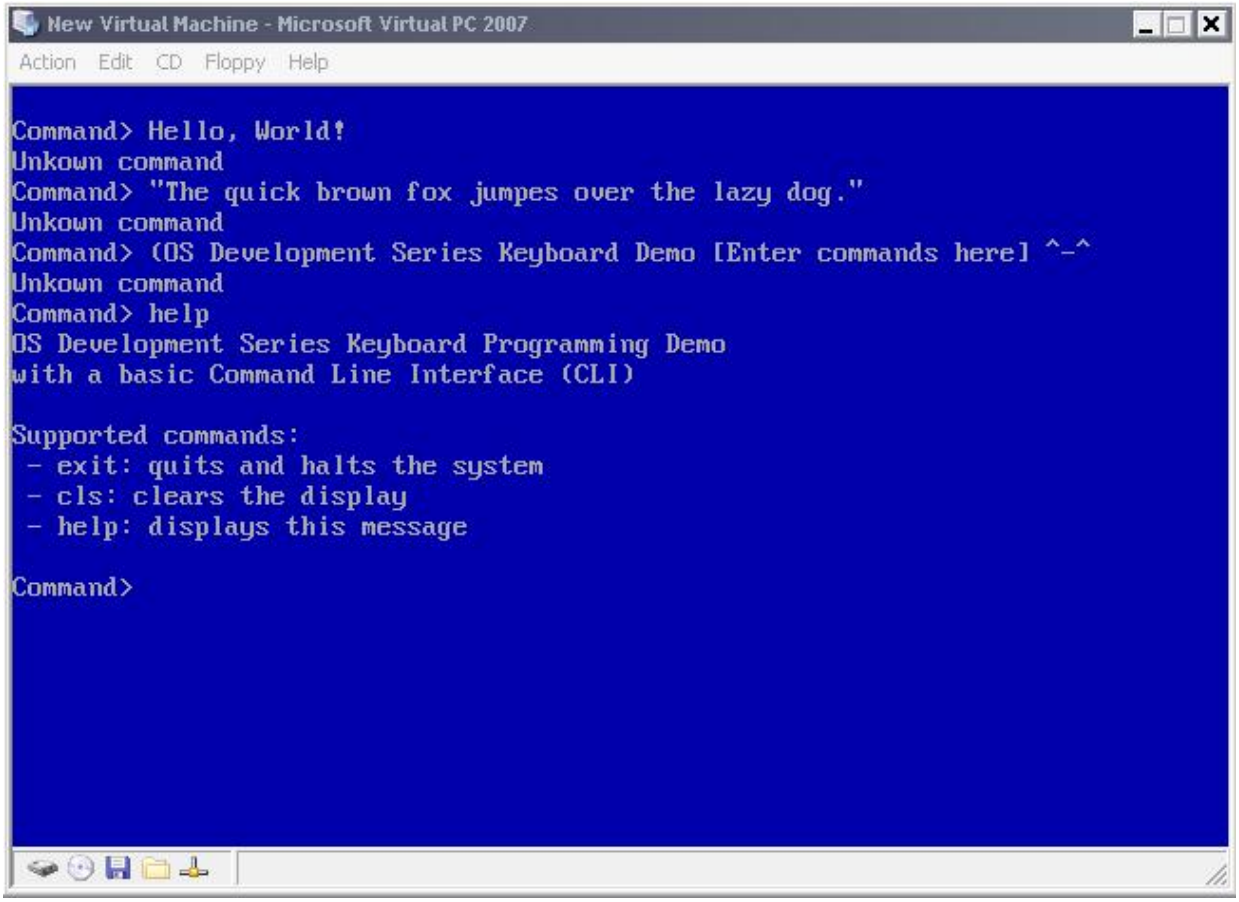
Command 0xFE - System Reset

Causes the controller to pulse bit 0 of the controllers output port (pin P0) which resets the CPU. This basically does the same thing as sending the **Write Output Port** command resetting bit 0. Send this command if you would like to reset the system in a nice way.

```
//! reset the system
void kkybrd_reset_system () {

        //! writes 11111110 to the output port (sets reset system line low)
        kybrd_ctrl_send_cmd (KYBRD_CTRL_CMD_WRITE_OUT_PORT);
        kybrd_enc_send_cmd (0xfe);
}
```

Keep in mind that this may not work on all systems. An easy way to see if it works or not is to see if your program is still executing after the above routine :)

Demo



The first interactive demo

DEMO DOWNLOAD

This is the most complex demo so far. It uses all of the code from the previous chapter along with an additional keyboard driver and basic **Command Line Interface (CLI)** to make things more interesting. Because of this, it is also our first interactive demo and can be extended upon with your own commands as well.

This demo also adds the **sleep ()** routine which is used to delay between each key read along with **scrollling** to the debug output console routines allowing the screen to scroll. Cool, huh?

I am not going to cover the code for the CLI itself as it is meant to be very simple. Rather, I want to focus on some of the more important parts of the demo.

Keyboard: Piecing it Together

We have looked at some of the routines from this demos keyboard driver already. We looked over communicating with the keyboard encoder and controller; and even routines for several different important functions for enabling, disabling, testing, LED updating, system reset, and more. This is great, but we are missing a few important details that tie everything together. Lets take a look, shall we?

Keyboard: Storing the current state

As you know, you can press any of the keys on your keyboard at any time. Because of this, there needs to be a way to scan each key to see if they are down or not. The good thing here is that the keyboard encoder already does this! To make things more easier, the keyboard encoder sends the scan code directly to the onboard keyboard controller which in turns invokes IRQ 1.

As long as IRQ 1 is not masked, we can install our own interrupt handler at IRQ 1 so that we can get notified whenever a scan code is sent from the keyboard encoder. What does this mean? **Our interrupt handler will be invoked any time a scan code is sent to the keyboard controller**. This can happen at any time.

Because of this, we need to somehow determin what the scan code is inside of the handler by polling the keyboard controller for it. However, we may want to do different things if a key is down (like the caps lock or num lock keys). These keys supposed to stay on or off when they are pressed. Then, what about other keys like shift? These keys must be held down and released when the key is released.

Because of this, we need to come up with a method of storing the current state of these keys and the last scan code read so that they can be retrieved again later after the IRQ has already returned. This can be done by

storing the current state in a few global varables or a structure and simply using them.

Keyboard: Interrupt Handling

This one is important. Remember that, for each key stroke and key release several bytes (The scan code) is sent to the keyboard controller? When this occurs, the keyboard controller signals the **Programmable Interrupt Controller (PIC)** to generate **IRQ 1**. Yes, dear readers, this in turn causes the PIC to execute our keyboard interrupt handler.

The purpose of the interrupt handler is to update the current state of the driver and to decipher the scan code by converting it to a format that can be used by the driver and the system. Yep, thats all that is to it ;)

The interrupt handler is what ties everything together. It is a little big so I decided to not put it in this text, however I urge everyone to take a look at it to see how it works.

Keyboard: Initialization

Remember that the keyboard controller is connected indirectly to the programmable interrupt IRQ 1 line? Because we mapped the IRQs using the PIC to start from interrupt vector 32 (IRQ 0), IRQ 1 is at interrupt vector 33. Because of this, we need to install our interrupt handler using our **setvect** routine to use interrupt vector 33.

Everything else is pretty simple. We simply clear out the current driver state (stored as globals) and clear the LEDs using our **kkybrd_set_leds** routine.

```
//! prepares driver for use
void kkybrd_install (int irq) {

	//! Install our interrupt handler (irq 1 uses interrupt 33)
	setvect (irq, i86_kybrd_irq);

	//! assume Basic Assurance test (BAT) test is good
	_kkybrd_bat_res = true;
	_scancode = 0;

	//! set lock keys and led lights
	_numlock = _scrolllock = _capslock = false;
	kkybrd_set_leds (false, false, false);

	//! shift, ctrl, and alt keys
	_shift = _alt = _ctrl = false;
}
```

Conclusion

Thats it for this chapter! The system is starting to get interesting, don't you think? We can possibly expand on this demo to make it actually useful to an extent, which is nice. However, we are rather limited in what we currently can do. Wouldn't it be useful if we can run other programs to do our work for us? Or even to load other files from disk? While abstracting a complete filesystem structure is very complex topic; we can, however, focus on loading files from a single disk.

However, we run into a problem. In order for us, at the minimum, be able to load a file from disk we have to program the **Floppy Disk Controller (FDC)** first. This is the topic for the next chapter. *Hope to see you there! :)*

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Operating Systems Development - FDC Programming

by Mike, 2009

This series is intended to demonstrate and teach operating system development from the ground up.

8272A Floppy Disk Controller

Introduction

Yey! Its finally time to work with the floppy drive! This chapter covers almost everything there is to know about the floppy drive and programming the floppy disk!

Here is what is on the menu for this chapter:

- FDC and FDD History
- Disk Layout
- CHS, LBA
- FDD Structure
- FDC Hardware
- Interfacing with the FDC
- FDC registers and commands

History

The **Floppy Disk Controller (FDC)** is the controller that interfaces with the **Floppy Disk Drive (FDD)**. The PC useually uses a form of the **NEC ?PD765 FDC**. PS/2 useally uses a form of the **Intel 82077A** while the AT useally uses a form of the **Intel 82072A** microcontroller.

The **Floppy disk drive (FDD)** is a device that is capable of reading and writing data to a **floppy disk**.

In 1971, David L. Noble, hired by Alan Shugart, who was the IBM Direct Access Storage Product Manager, tried to develop a new storage tape format for their System/370 mainframes. IBM was looking to create something that is smaller and faster then tape drives when reloading the microcode for their **Initial Control Program Load (ICPL)**. Nobles team worked on a product under the code name "Minnow" called a "memory disk". It was a read only, 8 inch diskette, having the capacity of 80 kilobytes. It was commercially released in 1971 and shipped with all System/370 mainframes.

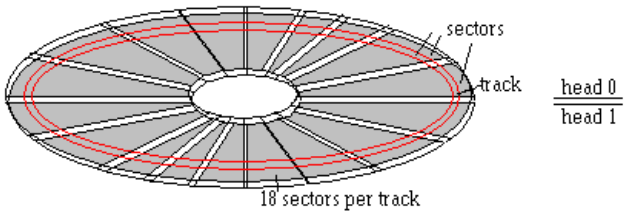
When Alan Shugart left IBM and moved to Memorex, his team shipped the Memorex 650 in 1972, the first read/write floppy disk drive.

Floppy disks were invented by IBM in 8 inch, 5 and 1/4 inch and 3 1/2 inch formats.

Disk Structure

Physical Layout

Understanding the disk structure is important. Here is the layout of a floppy disk:



This is the physical layout of a generic 3-1/2" floppy disk. Here, we are looking at Head 1 (The front side), and the Sector represents 512 bytes. A Track is a collection of sectors.

Note: Remember that 1 sector is 512 bytes, and there are 18 sectors per track on floppy disks.

Looking at the above picture, remember:

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

To better understand everything, we should have a look at **CHS**. Lets take a look at that next!

Cylinder / Head / Sector (CHS)

Sectors

A "Sector" simply represents a goupe of 512 bytes. So, Sector 1 represents the first 512 bytes of a disk.

Head

A "Head" (or Face) represents the side of the disk. Head 0 is the front side, Head 1 is the back side. Most disks only have 1 side, hence only 1 head ("Head 1")

Track

A track is one ring around the disk. In the case of floppy disks, 18 sectors span a single track.

The **Cylinder number represents a track number on a single disk**. In the case of a floppy disk, It represents the Track to read from.

There is 18 sectors per track. 80 tracks per side.

Understanding CHS

The floppy disk addresses using CHS format. In order to read or write from any location on disk, we must tell the FDC to move the **Read/Write Head** to the exact track, cylinder, and sector on the disk to read or write to.

Linear Block Addressing (LBA)

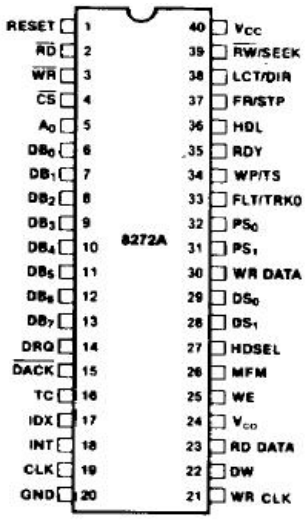
We can also provide a more abstract way of reading and writing to disks using **Linear Block Addressing (LBA)** instead. LBA allows us to be able to read or write to any sector on disk from sector 0-2880.

Floppy Interfacing

Software interfaces with the floppy disk drive by controlling it through a floppy disk controller. Do to differences in floppy disk controllers, I would like to focus on the original 8272A Floppy Disk Controller. The image at the beginning of this chapter shows a typical 8272A Integrated Circuit (IC) controller. This is the IC that we will look at here.

Detail: 82072A Floppy Microcontroller

The 8272A IC has 40 pins. Lets take a look at it here. While we will take a brief look at all 40 pins, we will not look at it in full detail here as that is when we cross into the electronics field.



Most of these pins are not very useful for programming the controller. Other pins are more important to understand, however. Lets take a look. For completness sake, we will look at all of the pins brefily. You will see that the FDC indirectly communicates with both the **Programmable Interrupt Controller (PIC)**, the system bus, as well as the **Direct Memory Access** controller.

- **RESET Pin** - places the FDC in an idle state. It drives all output lines low. The **Vcc** pin is a +5 V power input.
- **GND Pin** - is the ground pin.
- **CLK Pin** - typical Single Phase 8 MHz Squarewave clock signal.
- **RD Pin** - tells the FDC that the current operating is a read operation.
- **WR Pin** - is simular, but for a write operation.
  - These are set by the **Control Bus** in an **I/O read/write** operation by software.
- **CS Pin** - Chip Select
- **DB0 - DB7 Pins** - bidirectional 8 bit data bus. It connects indirectly to the systems primary **Data Bus**.
- **A0 Pin** - **Data/Status Register Select** pin. If it is high (1), it tells the FDC to place the contents in its **Data Register** to the data bus. If the line is low (0), it copies the contents of the **Status Register** to the data bus. This is done through the output data bus pins DB0 - DB7, which in turn is through the systems data bus which can be read by software.
- **DRQ Pin** - **Data Direct Memory Access (DMA) Request** pin. **If this line is high (1), the FDC is making a DMA request.**
- **DACK Pin** - **DMA Acknowledge** pin. When the controller is performing a DMA transfer, this line will be low (0).
- **TC Pin** - When the DMA transfer is completed, the FDC sets the **Terminal Count** pin, TC to high (1).
- **IDX Pin** - high when the FDC is at the beginning of a disk track.

- **INT Pin** - is high (1) when the FDC sends an **Interrupt Request (IR)**. This line is indirecty connected to the **IR6** on the **Programmable Interrupt Controller (PIC)**.
- **RW/Seek Pin** - Sets seek mode of read/write mode. 1: Seek mode, 0: Read/Write mode.
- **LCT/DIR Pin** - **Low current/Direction** pin.
- **FR/STP Pin** - **Fault reset/Step** pin.
- **HDL Pin** - **Head Load** pin.Command causes the Read/Write head in the FDD to contact the diskette
- **RDY Pin** - **Ready** pin. Indicates that the FDD is ready to send or recieve data
- **WP/TS Pin** - **Write protect/Two side** pin. In Read/Write mode, set high if media is write protected. If seek mode, set high if media is two sided.
- **FLT/TRK0 Pin** - **Fault/Track 0** pin. In Read/Write mode, set high on a detected FDD fault.
- **PS0 - PS2 Pins** - **Precompensation (Pre-shift)** pins. Write precompensation status during **MFM** mode.
- **WR DATA Pin** - Write data pin
- **RD DATA Pin** - Read data pin
- **DS0 - DS1 Pins** - Drive select pins
- **HDSEL Pin** - **Head Select Pin.** When high (1), the FDC sets the FDD to access Head 1. When low, it is head 0.
- **MFM Pin** - When high, FDC is in **MFM** mode. If low (0), it operates in **FM** mode.
- **WE Pin** - **Write enable pin.**
- **VCO Pin** - **VCO Sync pin.** When 0, inhibits **VCO** in **PLL**. When 1, enables **VCO**.
- **DW Pin** - Data Window pin. Generated by PLL, used for sample data from the FDD.
- **WR CLK Pin** - Write Clock

The FDC can operate with or without a **Direct Memory Access (DMA) controller**. If it is operating in a non DMA mode, it will generate **IRQ 6** for every transfer of a data byte between the processor and the FDC. In DMA mode, the processor will load a command into the FDC and all data transfers will occur under control of the FDC and DMA controllers.

This is important! You do not need to know all of the FDC pins. Rather, just remember that the **FDC communicates with three primary controllers.** The first is one of possibly four **Floppy Disk Drives (FDD)** internal controllers, the **programmable interrupt controller (PIC)**, and the **Direct Memory Access (DMA) controller**. Software communicates with the FDC by the processors standard **IN/OUT** port i/o instructions.

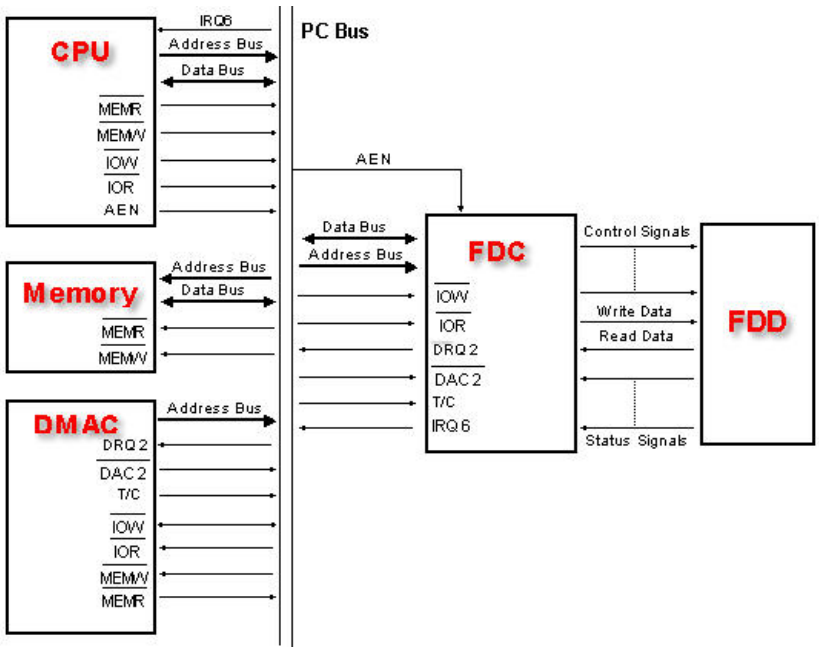
Several registers in the FDC are mapped into the processors i/o address space. As with standard I/O port reads, during an in and out operation, the processor sets the READ or WRITE line on the control bus, and the port address on the address bus. This is done on the **system bus** or the **Industry Standard Architecture (ISA)** bus.

On newer hardware, the FDC is not directly connected to the ISA bus, but is rather integrated as a Super I/O IC and communicates with the processor through the Super I/O's **Low Pin Count** bus.

Okay! We know how the software can communicate with the FDC. Where does the PIC and DMA come into play?

Looking at the pin listing above, we can see that the FDC has a pin called **INT**. This line is indirectly connected to the **Programmable Interrupt Controller IR 6** line. The FDC will pull this line high (1) whenever a byte of data is ready to be read or written. This also pulls the PIC IR 6 line high. From here, the PIC takes control. It masks out the other lines and determins if it can be services. It notifies the processor of an interrupt by activating the processors **Interrupt Acknowledge (INTA)** pin. After the processor verifies that it is safe to service the interrupt, it resets the INTA line to ackowledge the PIC to proceed. The PIC places the interrupt vector that this IRQ is mapped to use (set up during initializing the PIC). The processor takes the IRQ, gets its address from the idtr, and voila - our interrupt is called.

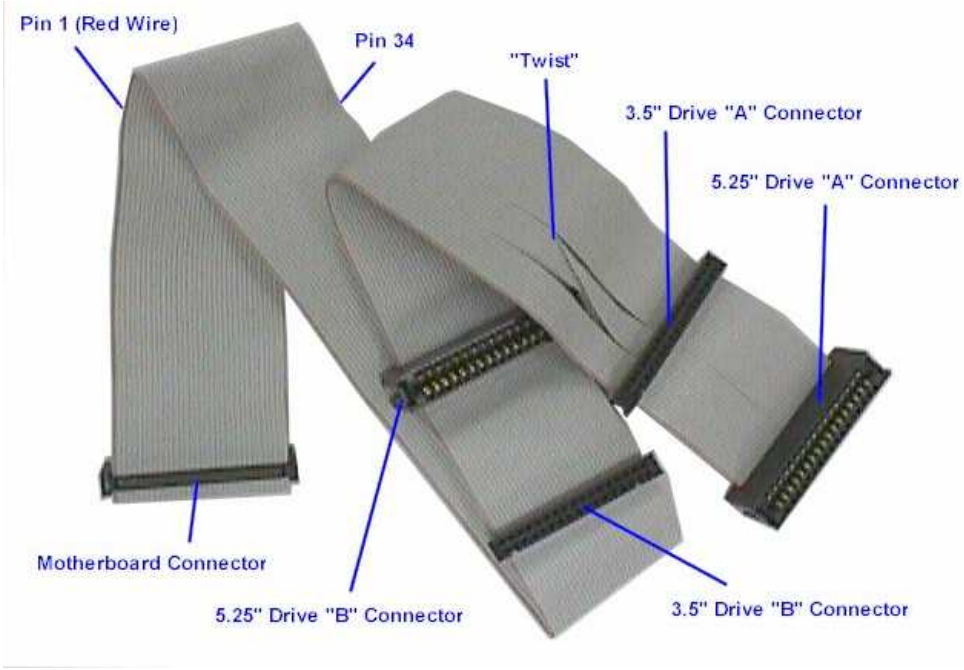
The FDC can also be programmed to operate in DMA mode. The DMA is a controller that we have not looked at yet. Because of this, I do not want to get too involved with it. However we may go over it in the next chapter for completness. **The FDC is connected to DMA channel 2**.



Thats all there is to it for the FDC hardware. **Their can be multiple FDCs inside of a computer system. Each FDC can connect up to 4 Floppy Disk Drives (FDDs). This is important!** Alot of times when communicating with a FDC, you have to select which FDD that the request is for.

Floppy Interface Cable

The FDC communicates with a FDD through a **Floppy Interface Cable**, which is a form of a **Parallel ATA (PATA)** cable also known as an **Integrated Drive Electronics (IDE)** cable which evolved from **Western Digital**.



You should notice a twist in the above cable. That will be described a little shortley. This cable has 40 pins. Through these 40 pins, the FDC can talk to different FDD's that are connected to the cable.

Some registers that are used to communicate with the FDC allow you to detect the input pins of the controller and the cable. Because of this, we should probably at least take a small glance at the 40 lines of the cable.

Floppy Interface Cable Pins			
Pin	Description	Pin	Descripton
0	Reset	20	DDRQ
1	Ground	21	Ground
2	Data pin 7	22	I/O Write
3	Data pin 8	23	Ground
4	Data pin 6	24	I/O Read
5	Data pin 9	25	Ground
6	Data pin 5	26	IOCHRDY
7	Data pin 10	27	Cable Select (CS)
8	Data pin 4	28	DDACK
9	Data pin 11	29	Ground
10	Data pin 3	30	Interrupt
11	Data pin 12	31	(No connection)
12	Data pin 2	32	Address 1
13	Data pin 13	33	GPIO_DMA66_Detect
14	Data pin 1	34	Address 0
15	Data pin 14	35	Address 2
16	Data pin 0	36	Chip Select 1
17	Data pin 15	37	Chip Select 3
18	Ground	38	Activity
19	Key or Vcc_in	39	Ground

- More to be added later -

FDC Programming

FDC Operating Modes

Most FDC's these days are more advanced then the original 8272 microcontroller. To acheive backward compatability, newer FDC's add additional pins to the controller and allow different registers to be communicated with when operating in a specific mode. For example, the **Status Register A** mode is only accesable when the controller is running in **PC-AT** mode. Upon controller reset, the controller operates in the default **82077A** mode.

Waiting for an IRQ

Remember that the FDC uses IRQ 6? The FDC will send a byte after the completion of a read or write command, or, depending on its mode, for every byte transferred. It will also send an IRQ when the controller is reset during initialization.

For our purposes, we will be operating the FDC in a DMA mode. Basically what this means is that we will only be getting an interrupt whenever a read, write, seek, or calibrate command completes as well as during initialization.

In all cases, however, this means that we will need to wait for an IRQ to fire so we know the command completes. A way for us to do this is to have the IRQ set a global when it fires, and provide an irq_wait like function that waits for the IRQ, and resets the global when it fires.

Lets do that now. First the IRQ:

```
const int FLOPPY_IRQ = 6;

//! set when IRQ fires
static volatile uint8_t _FloppyDiskIRQ = 0;

void _cdecl i86_flpy_irq () {

        _asm add esp, 12
        _asm pushad
        _asm cli

        //! irq fired
        _FloppyDiskIRQ = 1;

        //! tell hal we are done
        interruptdone( FLOPPY_IRQ );

        _asm sti
        _asm popad
        _asm iretd
}
```

This looks as simple as the IRQ in the PIT, doesnt it? :) Oh, right, and now we wait:

```
//! wait for irq to fire
inline void flpydsk_wait_irq () {

        //! wait for irq to fire
        while ( _FloppyDiskIRQ == 0)
                ;
        _FloppyDiskIRQ = 0;
}
```

Simple enough. So, assuming we send a command, like a read or write command, just call **flpydsk_wait_irq()**. When it completes, you know the command finished and its safe to continue. Cool, huh? ;)

DMA?

What? We are programming the FDC in DMA mode? But we have not covered the DMA yet! Yes, yes indeed this poses a problem.

I was originally going to program the FDC in Non-DMA mode. However, while this might work in some cases, alot of emulators and even some hardware do not support it anymore. Because of this, to retain portability, I decided that the best bet is to stick with using the DMA (**Direct Memory Access Controller [DMAC]**).

However, because we have not covered the DMA yet in detail, we run into a problem. I figure, rather then throwing a whole DMA interface to you without explination, we can just hack together three basic DMA routines and rewrite them more throughley later ;)

flpydsk_initialize_dma basically creates a buffer for the DMA to use at physical address 0x1000 - 0x10000 (64k). When we read a sector from disk, the DMA will put the sector data to this location so please be sure that nothing is there as it will be overwritten. You can choose another location if you like, however there are some rules:

- The buffer cannot cross 64k boundaries. It should stay at a 64k boundery for best performance
- The area of memory it writes to must be idenitity mapped or its frame address mapped to a page. The DMA **always** works with physical memory

The demo uses 0x1000 + 64k for the buffer so you should keep it there if you dont feel confortable changing it.

dma_read and **dma_write** just tells the DMA to start reading or writing the data that the FDC sends it. This will be the sector that we tell the FDC to read or write. For example, if we tell the FDC to read a sector, it will give the sector data to the DMA to be placed in the buffer that we set it to (which is at 0x1000). Cool, huh?

```
//! initialize DMA to use phys addr 1k-64k
void flpydsk_initialize_dma () {

        outportb (0x0a,0x06);   //mask dma channel 2
        outportb (0xd8,0xff);   //reset master flip-flop
        outportb (0x04, 0);     //address=0x1000
        outportb (0x04, 0x10);
        outportb (0xd8, 0xff);  //reset master flip-flop
        outportb (0x05, 0xff);  //count to 0x23ff (number of bytes in a 3.5" floppy disk track)
        outportb (0x05, 0x23);
        outportb (0x80, 0);     //external page register = 0
        outportb (0x0a, 0x02);  //unmask dma channel 2
}

//! prepare the DMA for read transfer
void flpydsk_dma_read () {

        outportb (0x0a, 0x06); //mask dma channel 2
        outportb (0x0b, 0x56); //single transfer, address increment, autoinit, read, channel 2
```

```
        outportb (0x0a, 0x02); //unmask dma channel 2
}

//! prepare the DMA for write transfer
void flpydsk_dma_write () {

        outportb (0x0a, 0x06); //mask dma channel 2
        outportb (0x0b, 0x5a); //single transfer, address increment, autoinit, write, channel 2
        outportb (0x0a, 0x02); //unmask dma channel 2
}
```

If you dont understand the above code, dont worry. Everything reguarding the DMA will be rewritten and explained in the next tutorial when we cover the DMA in more detail.

FDC Port mapping

The FDC has four external registers that are mapped into the i86 I/O address space. These can be accessed by software through standard I/O instructions. I **bolded** these registers.

Some systems may provide more external registers to their FDC's then the primary four.

The second FDC is typically mapped to I/O ports 0x370 - 0x377.

Because there are two sets of ports for two different FDC's, this table will include both port sets.

Floppy Disk Controller Ports			
Port (FDC 0)	Port (FDC 1)	Read/Write	Descripton
Primary FDC Registers			
0x3F2	0x372	Write Only	Digital Output Register (DOR)
0x3F4	0x374	Read Only	Main Status Register (MSR)
0x3F5	0x375	Read / Write	Data Register
0x3F7	0x377	Read Only	AT only. Configuation Control Register (CCR)
0x3F7	0x377	Write Only	AT only. Digital Input Register (DIR)
Other FDC Registers			
0x3F0	0x370	Read Only	PS/2 only. Status Register A (SRA)
0x3F1	0x371	Read Only	PS/2 only. Status Register B (SRB)
0x3F4	0x374	Write Only	PS/2 only. Data Rate Select Register (DSR)

We will take a look at the registers closer - bit by bit - in the next section. Well, the important ones anyways. I may decide to update this chapter covering the other registers for completness purposes, though. For now, we will only focus on the first four registers shown above.

Remember that all of this code is in the demo at the end of this chapter.

```
enum FLPYDSK_IO {

        FLPYDSK_DOR             =       0x3f2,
        FLPYDSK_MSR             =       0x3f4,
        FLPYDSK_FIFO            =       0x3f5,  //data register
        FLPYDSK_CTRL            =       0x3f7
};
```

Registers

Status Register A (SRA) (PS2 Mode Only)

You do not need to know this register. It is here for completness only.

This is a read only register that monitors the state of several interface pins on the controller. It is not accessable when the controller is in PC-AT Mode. This is a read only register.

The exact format of this register may depend on the model of the controller.

- **Bit 0** DIR
- **Bit 1** WP
- **Bit 2** INDX
- **Bit 3** HDSEL
- **Bit 4** TRKO
- **Bit 5** STEP Flip/Flop
- **Bit 6** DRV2
- **Bit 7** INTERRUPT line state (interrupt pending)

Warning: These bits can change between controller models.

Do not worry if this register seems complex; it can be without experience in electronics. It is here for completeness only and will not be used in the series.

Status Register B (SRB) (PS/2 Mode Only)

You do not need to know this register. It is here for completness only.

Simular to the above register, this allows us to monitor the state of several lines of the FDC. It is not accessable when the FDC is in PC-AT Mode. This is a read only register.

- **Bit 0** MOT EN0 (Motor Enable 0)
- **Bit 1** MOT EN1 (Motor Enable 1)
- **Bit 2** WE Flip/Flop
- **Bit 3** Read Data (RDDATA) Flip/Flop
- **Bit 4** Write Data (WRDATA) Flip/Flop
- **Bit 5** Drive Select 0
- **Bit 6** Undefined; Always 1
- **Bit 7** Undefined; Always 1

Warning: These bits can change between controller models.

Do not worry if this register seems complex; it can be without experience in electronics. It is here for completeness only and will not be used in the series.

Data Rate Select Register (DSR)

You do not need to know this register. It is here for completness only.

This is a write only register that allow you to change the timings of the drive control signals. It is used by writing to I/O port 0x3f4 (FDC 0) or 0x374 (FDC 1).

This is an 8 bit register. It has the following format:

- **Bit 0** DRATE SEL0
- **Bit 1** DRATE SEL1
- **Bit 2** PRE-COMP 0
- **Bit 3** PRE-COMP 1
- **Bit 4** PRE-COMP 2
- **Bit 5** Must be 0
- **Bit 6** POWER DOWN: Deactivates internal clocks and shuts off the internal oscillator
- **Bit 7** S/W RESET: Reset the internal oscillator

PRE-COMP 0 - PRE-COMP 2 are a little complex. These adjusts the **WRDATA** output pins for the **bit shifting** that can occur on magnetic media, such as floppy drives. To adjust the precompensation delay, we can set these bits to one of the following:

- **000** Default (250-500 Kbps, 125 ns. 1 Mbps, 41.67 ns)
- **110** 250 ns
- **101** 208.33 ns
- **100** 166.67 ns
- **011** 125 ns
- **010** 83.34 ns
- **001** 41.67 ns
- **111** Disabled

DRATE SEL0 - DERATE SEL 1 are used to set the data rate. Valid values are shown below.

- **00** 500 Kbps
- **10** 250 Kbps
- **01** 300 Kbps
- **11** 1 Mbps

Warning: Setting Data Rates greater then drive can handle may cause errors.

Digital Output Register (DOR)

Yey! The first useful register! **This one is important to know**.

This is a **write only** register that allows you to control different functions of the FDC, such as the FDD motor control, operation mode (DMA and IRQ), reset, and drive. It has the format:

- **Bits 0-1** DR1, DR2
  - 00 - Drive 0
  - 01 - Drive 1
  - 10 - Drive 2
  - 11 - Drive 3
- **Bit 2** REST
  - 0 - Reset controller
  - 1 - Controller enabled
- **Bit 3** Mode
  - 0 - IRQ channel
  - 1 - DMA mode
- **Bits 4 - 7** Motor Control (Drives 0 - 3)
  - 0 - Stop Motor for drive
  - 1 - Start Motor for drive

This is an easy one! Basically when sending a command to control the functionality of the FDC, just build up a bit pattern to select what drive this is for (Remember that a single FDC can communicate with four FDD's!), the controller reset status, mode of operation (Remeber that the FDC can operate in both DMA and IRQ modes?) and the status of that particular FDD internal motor.

Here is an example. Lets say we want to start up the motor for the first floppy drive (FDD 0). **Starting the motor for the FDD is needed before performing any read or write operations to it!** To start it, just set the bit (4-7) that corrosponds to the drive you want to start or stop the motor. Keeping all other bits at 0 will be a normal operation (IRQ mode, reset controller.) Knowing that the DOR is mapped to the processors i/o address space at port 0x3f2, this becomes very simple. First, we will create bit masks for the register to increase readability. Rememeber that all of this code is also in the demo at the end of this tutorial.

```
enum FLPYDSK_DOR_MASK {

        FLPYDSK_DOR_MASK_DRIVE0                 =       0,      //00000000      = here for completeness sake
        FLPYDSK_DOR_MASK_DRIVE1                 =       1,      //00000001
        FLPYDSK_DOR_MASK_DRIVE2                 =       2,      //00000010
        FLPYDSK_DOR_MASK_DRIVE3                 =       3,      //00000011
        FLPYDSK_DOR_MASK_RESET                  =       4,      //00000100
        FLPYDSK_DOR_MASK_DMA                    =       8,      //00001000
        FLPYDSK_DOR_MASK_DRIVE0_MOTOR           =       16,     //00010000
        FLPYDSK_DOR_MASK_DRIVE1_MOTOR           =       32,     //00100000
        FLPYDSK_DOR_MASK_DRIVE2_MOTOR           =       64,     //01000000
        FLPYDSK_DOR_MASK_DRIVE3_MOTOR           =       128     //10000000
};
```

Using the above bit masks, we can just bitwise OR the different bits that we would like to set. So, to start the motor for floppy drive 0:

```
outportb (FLPYDSK_DOR, FLPYDSK_DOR_MASK_DRIVE0_MOTOR | FLPYDSK_DOR_MASK_RESET);
```

Remember that FLPYDSK_DOR was defined ealier as 0x3f2, which is the i/o address of the DOR FDC register. The above also resets the controller.

To turn this same motor off, just send the same command but without the motor bit set:

```
outportb (FLPYDSK_DOR, FLPYDSK_DOR_MASK_RESET);
```

Warning: Give the motor some time to start up! Remember that the internal FDD motor is mechanical. Like all mechanical devices, they tend to be slower then the speed of the running software. Because of this, whenever starting up a FDD motor, always give it a little time to spin up before attempting to read or write to it.

The DOR is a write only register. To inforce this, lets create a routine for it:

```
void flpydsk_write_dor (uint8_t val ) {

        //! write the digital output register
        outportb (FLPYDSK_DOR, val);
}
```

Lets move on to the next important register!

Main Status Register (MSR)

The **Main Status Register (MSR)** follows a *gasp!* specific bit format! Bet you did not see that one coming! Okay, okay, lets get back on track here (pun intended). Here is the format of the MSR:

- **Bit 0** - FDD 0: 1 if FDD is busy in seek mode
- **Bit 1** - FDD 1: 1 if FDD is busy in seek mode
- **Bit 2** - FDD 2: 1 if FDD is busy in seek mode
- **Bit 3** - FDD 3: 1 if FDD is busy in seek mode
  - 0: The selected FDD is not busy
  - 1: The selected FDD is busy
- **Bit 4** - FDC Busy; Read or Write command in progress
  - 0: Not busy
  - 1: Busy
- **Bit 5** - FDC in Non DMA mode
  - 0: FDC in DMA mode
  - 1: FDC not in DMA mode
- **Bit 6** - DIO: direction of data transfer between the FDC IC and the CPU
  - 0: FDC expecting data from CPU
  - 1: FDC has data for CPU
- **Bit 7** - RQM: Data register is ready for data transfer
  - 0: Data register not ready
  - 1: Data register ready

This MSR is a simple one. It containes the current status information for the FDC and disk drives. Before sending a command or reading from the FDD, we will need to always check the current status of the FDC to insure it is ready.

Here is an example of reading from this MSR to see if its busy. We first define the bit masks that will be used in the code. Notice how it follows the format shown above.

```
enum FLPYDSK_MSR_MASK {

        FLPYDSK_MSR_MASK_DRIVE1_POS_MODE        =       1,      //00000001
        FLPYDSK_MSR_MASK_DRIVE2_POS_MODE        =       2,      //00000010
        FLPYDSK_MSR_MASK_DRIVE3_POS_MODE        =       4,      //00000100
        FLPYDSK_MSR_MASK_DRIVE4_POS_MODE        =       8,      //00001000
        FLPYDSK_MSR_MASK_BUSY                   =       16,     //00010000
        FLPYDSK_MSR_MASK_DMA                    =       32,     //00100000
        FLPYDSK_MSR_MASK_DATAIO                 =       64,     //01000000
        FLPYDSK_MSR_MASK_DATAREG                =       128     //10000000
};
```

Easy, huh? So lets test if the FDC is busy (BUSY flag is set.) Knowing that FLPYDSR_MSR is 0x3f4, the i/o port address for the MSR, all we need to do is this:

```
if ( inportb (FLPYDSK_MSR) & FLPYDSK_MSR_MASK_BUSY )
        //! FDC is busy
```

When sending a read or write command, all we need to do is wait until this bit is 0. Cool, huh?

To make readability easier, I decided to hide this in a routine so here it is. This routine just returns the status of the FDC.

```
uint8_t flpydsk_read_status () {

        //! just return main status register
        return inportb (FLPYDSK_MSR);
}
```

Tape Drive Register (TDR)

You do not need to know this register. It is here for completness only.

This register allows us to assign tape drive support to a specific drive during initialization of that drive. This is a read/write register and is 8 bits in size. However only the first two bits are defined. They both are used to select between drive 0 - 3. Selecting Drive 0 is not allowed as that is reserved for the floppy boot device. Because of this, it is not in the bit list shown below.

- 00: None.
- 01: Drive 1
- 10: Drive 2
- 11: Drive 3

Only a hardware reset will reset this register. A software reset has no effect. Do not worry if you dont know much about tape drives - this register does not apply to us and will not be used in the series. It is here only for completeness. :)

Data Register

This is a 8 or 16 bit read/write register. The actual size of the register is specific on the type of controller. **All command paramaters and disk data transfers are read to and written from the data register.** This register does not follow a specific bit format and is used for generic data. It is accessed through I/O port 0x3f5 (FDC 0) or 0x375 (FDC 1).

Note: Before reading or writing this register, you should always insure it is valid by first reading its status in the Master Status Register (MSR).

Remember: All command bytes and command paramaters are sent to the FDC through this register! You will see examples of this in the command section below, so dont worry to much about it yet.

If an invalid command was issued, the value returned from the data register is 0x80.

The following routines read from this register and are use in the demo. It attemps to wait until the data register is safe to read or write to, then it either reads it (read_data function) or write it (send_command function).

```
void flpydsk_send_command (uint8_t cmd) {

        //! wait until data register is ready. We send commands to the data register
        for (int i = 0; i < 500; i++ )
                if ( flpydsk_read_status () & FLPYDSK_MSR_MASK_DATAREG )
                        return outportb (FLPYDSK_FIFO, cmd);
}

uint8_t flpydsk_read_data () {

        //! same as above function but returns data register for reading
        for (int i = 0; i < 500; i++ )
                if ( flpydsk_read_status () & FLPYDSK_MSR_MASK_DATAREG )
                        return inportb (FLPYDSK_FIFO);
}
```

Digital Input Register (DIR)

You do not need to know this register. It is here for completness only.

Okay, there was a digital output register (DOR) so I am sure you seen this one coming :) This is a read only register in all operation modes of the controller. Only bit 7 is defined when running in PC-AT Mode, all other bits are undefined and should not be used. In other operation modes, Bit 7 is undefined.

Bit 7 (DSK CHG) monitors the DSK CHG pin of the FDC. Looking at our pin layout at the beginning of this chapter, you will see that there is no DSK CHG pin. This has to do with the differences between the newer FDC models and the original model. Newer models added and changed different bits in this register to monitor newer pins on the FDC, such as DMA GATE, DRATE SEL0/1, etc. The values of this register is specific to the operation mode of the FDC.

Note that the bits in this register can change between models.

Configuation Control Register (CCR)

In PC/AT Mode, this register is known as the **Data Rate Select Register (DSR)** and only has the first two bits set (Bit 0=DRATE SEL0, Bit 1=DRATE SEL1.) This was listed in a table in the DSR register section. Lets take another look...

- **00** 500 Kbps
- **10** 250 Kbps
- **01** 300 Kbps
- **11** 1 Mbps

Bit 2 is NOPREC in Model 30/CCR modes and has no function. Other bits are undefined and may change depending on controller.

Like the other registers, I created a routine so we can write to this register.

```
void flpydsk_write_ccr (uint8_t val) {

        //! write the configuation control
        outportb (FLPYDSK_CTRL, val);
}
```

Commands

Abstract

Commands are used to control a FDD connected to the FDC for different operations, like reading and writing. They are written to the data register over the data bus (D0-D7) pins during a write operation (IO and WRITE control lines are set on the control bus.) In other words, a OUT assembly language instruction to the data register at port 0x3f5 (FDC 0) or 0x375 (FDC 1.)

Warning: Before sending a command or paramamter byte, insure the data register is ready to recieve data by testing bit 7 of the Main Status Register (MSR) first.

There are **thirteen** (or more depending on controller) commands. Each command can be 1 to 9 bytes in size. The FDC knows how many bytes to expect from the first command byte. That is, the first byte is the actual command that tells the FDC what we want it to do. The FDC knows how many more bytes to expect from this command (The command paramaters.)

Commands will only operate on a single head of the track. If you want to operate on both heads, you need to set the **Multiple Track Bit**. Alot of these commands follow bit formats (Will be shown below). This is where things get complicated.

A command byte only uses the low 4 bits of the byte for the actual command (can be more.) The high bits of these command bytes are for optional settings for the command. I call these **extended command bits** but it does not have an official name. There is a couple of these bits that are common for alot of the commands that we will need to use. We will look at these bits in the command byte later.

Okay, first lets take a look the command listing. We will then look at each one separately. Notice how they all only use the first 4 bits of the command byte.

```
enum FLPYDSK_CMD {

        FDC_CMD_READ_TRACK      =       2,
        FDC_CMD_SPECIFY         =       3,
        FDC_CMD_CHECK_STAT      =       4,
        FDC_CMD_WRITE_SECT      =       5,
        FDC_CMD_READ_SECT       =       6,
        FDC_CMD_CALIBRATE       =       7,
        FDC_CMD_CHECK_INT       =       8,
        FDC_CMD_WRITE_DEL_S     =       9,
        FDC_CMD_READ_ID_S       =       0xa,
        FDC_CMD_READ_DEL_S      =       0xc,
        FDC_CMD_FORMAT_TRACK    =       0xd,
        FDC_CMD_SEEK            =       0xf
};
```

To send a command to the FDC, remember that we have to write it to the data register, aka the FIFO. To do this, we first need to wait until the data register is ready by checking the bit in the MSR. Assuming **flpydsk_read_status ()** just returns the value from the MSR, lets hide all of this inside of a simpler method:

```
void flpydsk_send_command (uint8_t cmd) {

        //! wait until data register is ready. We send commands to the data register
        for (int i = 0; i < 500; i++ )
                if ( flpydsk_read_status () & FLPYDSK_MSR_MASK_DATAREG )
                        return outportb (FLPYDSK_FIFO, cmd);
}
```

Extended Command Bits

Some of these commands require you to pass several bytes before the command is executed. Others return several bytes. To make things easier to read, I have listed all of the commands, formats, and paramater bytes in tables. Each command comes with an explination and an example routine.

Okay, now remember when we mentioned exteneded command bits and how the commands above are only four bits? The upper four bits can be used for different things and purposes.

When describing the format of a command, we represent an extended bit with a character (like M or F.) For example, the Write Sector command has the format M F 0 0 0 1 1 0, where the first four bits (0 1 1 0) are the command byte and the top four bits, M F 0 0 represent different settings. M is set for multitrack, F to select what density mode to operate in for the command.

Here is a list of common bits:

- M - MultiTrack Operation

    - 0: Operate on one track of the cylinder
    - 1: Operate on both tracks of the cylinder
- F - FM/MFM Mode Setting
    - 0: Operate in FM (Single Density) mode
    - 1: Operate in MFM (Double Density) mode
- S - Skip Mode Setting
    - 0: Do not skip deleted data address marks
    - 1: Skip deleted data address marks
- HD - Head Number
- DR0 - DR1 - Drive Number Bits (2 bits for up to 4 drives)

The M, F, and S bits are very common to alot of the commands, so I decided to stick them in a nice enumeration. To set them, just bitwise OR these settings with the command that you would like to use.

```
enum FLPYDSK_CMD_EXT {

        FDC_CMD_EXT_SKIP        =       0x20,   //00100000
        FDC_CMD_EXT_DENSITY     =       0x40,   //01000000
        FDC_CMD_EXT_MULTITRACK  =       0x80    //10000000
};
```

GAP 3

GAP 3 referrs to the space between sectors on the physical disk. It is a type of **GPL (Gap Length)**.

```
enum FLPYDSK_GAP3_LENGTH {

        FLPYDSK_GAP3_LENGTH_STD = 42,
        FLPYDSK_GAP3_LENGTH_5_14= 32,
        FLPYDSK_GAP3_LENGTH_3_5= 27
};
```

Some commands require us to pass the GAP 3 code, so there it is :)

Bytes Per Sector

Some commands require us to pass in the bytes per sector. These cannot be any size, however, and always follows a formula:

```
2^n * 128, where ^ denotes "to the power of"
```

n is a number from 0-7. It cannot go higher then 7, as 2^7 * 128 = 16384 (16 kbytes). It is possible to select up to 16 Kbytes per sector on the FDC. Most drives may not support it, however.

Our list has the most common:

```
enum FLPYDSK_SECTOR_DTL {

        FLPYDSK_SECTOR_DTL_128  =       0,
        FLPYDSK_SECTOR_DTL_256  =       1,
        FLPYDSK_SECTOR_DTL_512  =       2,
        FLPYDSK_SECTOR_DTL_1024 =       4
};
```

...So, if a command requires us to pass the number of bytes per sector, dont put 512! rather, put FLPYDSK_SECTOR_DTL_512, which is 2.

How to pass paramaters to commands

If you recall, alot of commands require us to pass paramaters to it. To pass the paramaters, simply send them the same way the command was sent. For example, the specify command requires us to pass two paramaters to it. The command wont start without it so...

```
flpydsk_send_command (FDC_CMD_SPECIFY);
flpydsk_send_command (data);
flpydsk_send_command (data2);
```

Thats all there is to it ;)

How to get return values from commands

Unlike functions in programming in which you can ignore return values, the FDC requires for them to be processed in some way. Granted, you can still ignore them, but you must get them from the FDC. The FDC wont allow any more commands until it is done.

If the command returns data, it will be returned -- one at a time -- in the FIFO (Data register). So, to read them, you must continually read the FIFO to get all of the returned data.

Note: If a command returns data, it will send an interrupt that you must wait for. This is how you will know when the command is done and that it is safe to read the return values from the FIFO.

A good example of return values is the read sectors command. It requires us to wait for an IRQ so we know it completes, and returns 7 bytes. So to read all of the returned data bytes, we have to read from the data register one at a time:

```
for (int j=0; j<7; j++)
        flpydsk_read_data ();
```

Of course, for error checking purposes you should actually check some of the return values.

Write Sector

- Format: M F 0 0 0 1 1 0
- Paramaters:
  - x x x x x HD DR DR0
  - Cylinder
  - Head
  - Sector Number
  - Sector Size
  - Track Length
  - Length of GAP3
  - Data Length
- Return:
  - Return byte 0: ST0
  - Return byte 1: ST1
  - Return byte 2: ST2
  - Return byte 3: Current cylinder
  - Return byte 4: Current head
  - Return byte 5: Sector number
  - Return byte 6: Sector size

This command reads a sector from a FDD. For every byte in the sector, the FDC issues interrupt 6 and places the byte read from the disk into the data register so that we can read it in.

Read Sector

- Format: M F S 0 0 1 1 0
- Paramaters:
  - x x x x x HD DR1 DR0 = HD=head DR0/DR1=Disk
  - Cylinder
  - Head
  - Sector Number
  - Sector Size
  - Track Length
  - Length of GAP3
  - Data Length
- Return:
  - Return byte 0: ST0
  - Return byte 1: ST1
  - Return byte 2: ST2
  - Return byte 3: Current cylinder
  - Return byte 4: Current head
  - Return byte 5: Sector number
  - Return byte 6: Sector size

This command reads a sector from a FDD. For every byte in the sector, the FDC issues interrupt 6 and places the byte read from the disk into the data register so that we can read it in.

The following is the routine used in the demo. It first sets up the DMA to prepare for a read operation. It then executes the read sector command (FDC_CMD_READ_SECT) setting the commands M, F, and S bits with it (Multitrack read, double density, skip deleted address marks. Please see above for a list of all of these.)

Afterwords, it passes all of the commands paramaters to it to begin the read command. The sector size paramater is FLPYDSK_SECTOR_DTL_512 (bytes per sector), which, if you recall, is the value 2 (Please see the above **Bytes per sector** section for details.) Next paramater is the sectors per track (18). The next paramater is the GAP 3 length. We pass the value of the standard 3-1/2" floppy disk GAP 3 length (FLPYDSK_GAP3_LENGTH_3_5, which is 27).

The **Data Length** paramater byte is only valid if the sector size is 0. Else, it should be 0xff.

Because this command sends an IRQ after completion, we need to wait for the IRQ.

```
void flpydsk_read_sector_imp (uint8_t head, uint8_t track, uint8_t sector) {

        uint32_t st0, cyl;

        //! set the DMA for read transfer
        flpydsk_dma_read ();

        //! read in a sector
        flpydsk_send_command (
                FDC_CMD_READ_SECT | FDC_CMD_EXT_MULTITRACK |
                FDC_CMD_EXT_SKIP | FDC_CMD_EXT_DENSITY);
        flpydsk_send_command ( head << 2 | _CurrentDrive );
        flpydsk_send_command ( track);
        flpydsk_send_command ( head);
        flpydsk_send_command ( sector);
        flpydsk_send_command ( FLPYDSK_SECTOR_DTL_512 );
        flpydsk_send_command (
                ( ( sector + 1 ) >= FLPY_SECTORS_PER_TRACK )
                        ? FLPY_SECTORS_PER_TRACK : sector + 1 );
        flpydsk_send_command ( FLPYDSK_GAP3_LENGTH_3_5 );
```

```
        flpydsk_send_command ( 0xff );

        //! wait for irq
        flpydsk_wait_irq ();

        //! read status info
        for (int j=0; j<7; j++)
                flpydsk_read_data ();

        //! let FDC know we handled interrupt
        flpydsk_check_int (&st0,&cyl);
}
```

...After the IRQ fires, we read in all 7 return bytes. Then we send a SENSE_INTERRUPT command with flpydsk_check_int () which tells the FDC that we have handled the interrupt. (Please see the **Check Interrupt Status** section below.)

Wait... Where is the data at? Looking at the above command, we dont tell the FDC will to put the data at. This poses an interesting problem, dont you think?

Depending on the FDCs mode of operation, in Non-DMA mode, it will fire IRQ 6 for every byte. The byte of data read from disk is in the FIFO. In DMA mode (where we are in), it will give the data to the DMA, which will put the data into a buffer (wherever location we told the DMA to put it at.)

So, in our case, we set up the DMA buffer to 0x1000, remember? After calling the above routine, the sector data will be at 0x1000! Cool, huh? We can change its location by giving the DMA a different address.

Fix Drive Data / Specify

- Format: 0 0 0 0 0 0 1 1
- Paramaters:
  - S S S S H H H H - S=Step Rate H=Head Unload Time
  - H H H H H H H NDM - H=Head Load Time NDM=0 (DMA Mode) or 1 (DMA Mode)
- Return: None

This command is used to pass controlling information to the FDC about the mechanical drive connected to it. To make working with this command easier, lets write a routine for it:

```
void flpydsk_drive_data (uint32_t stepr, uint32_t loadt, uint32_t unloadt, bool dma ) {

        uint32_t data = 0;

        flpydsk_send_command (FDC_CMD_SPECIFY);

        data = ( (stepr & 0xf) << 4) | (unloadt & 0xf);
        flpydsk_send_command (data);

        data = (loadt) << 1 | (dma==true) ? 1 : 0;
        flpydsk_send_command (data);
}
```

Check Status

- Format: 0 0 0 0 0 1 0 0
- Paramaters:
  - x x x x x HD DR1 DR0
- Return:
  - Byte 0: Status Register 3 (ST3)

This command returns the drive status.

Calibrate Drive

- Format: 0 0 0 0 0 1 1 1
- Paramaters:
  - x x x x x 0 DR1 DR0
- Return: None

This command is used to position the read/write head to cylinder 0. After completion, the FDC issues an interrupt. If the disk has more then 80 tracks, you may need to issue this command several times. After issuing this command, always check to insure it is on the right track (**Check Interrupt Status** command.)

If, after the command, we are not on cylinder 0 yet, we issue the command again. When we find cylinder 0, we turn the motor off and return success. If we dont fine it after 10 tries we bail.

Note that we have to insure that the motor is running during this command. Also notice how we use the SENSE_INTERRUPT command (the flpydsk_check_int () call) to abtain the current cylinder.

```
int flpydsk_calibrate (uint32_t drive) {

        uint32_t st0, cyl;

        if (drive >= 4)
                return -2;

        //! turn on the motor
```

```
        flpydsk_control_motor (true);

        for (int i = 0; i < 10; i++) {

                //! send command
                flpydsk_send_command ( FDC_CMD_CALIBRATE );
                flpydsk_send_command ( drive );
                flpydsk_wait_irq ();
                flpydsk_check_int ( &st0, &cyl);

                //! did we fine cylinder 0? if so, we are done
                if (!cyl) {

                        flpydsk_control_motor (false);
                        return 0;
                }
        }

        flpydsk_control_motor (false);
        return -1;
}
```

Check Interrupt Status

- Format: 0 0 0 0 1 0 0 0
- Paramaters: None
- Return:
  - Byte 0: Status Register 0 (ST0)
  - Byte 1: Current Cylinder

This command is used to check information on the state of the FDC when an interrupt returnes.

```
void flpydsk_check_int (uint32_t* st0, uint32_t* cyl) {

        flpydsk_send_command (FDC_CMD_CHECK_INT);

        *st0 = flpydsk_read_data ();
        *cyl = flpydsk_read_data ();
}
```

Seek / Park Head

- Format: 0 0 0 0 1 1 1 1
- Paramaters:
  - x x x x x HD DR1 DR0 - HD=Head DR1/DR0 = drive
  - Cylinder
- Return: None

This command is used to move the read/write head to a specific cylinder. Simular to the calibrate command, we may need to send the command multiple times. Notice the call to check_int () to get the current cylinder after every attempt. We then test if the current cylinder is the cylinder we are looking for. If it is not, we try again. If it is, we return success.

```
int flpydsk_seek ( uint32_t cyl, uint32_t head ) {

        uint32_t st0, cyl0;

        if (_CurrentDrive >= 4)
                return -1;

        for (int i = 0; i < 10; i++ ) {

                //! send the command
                flpydsk_send_command (FDC_CMD_SEEK);
                flpydsk_send_command ( (head) << 2 | _CurrentDrive);
                flpydsk_send_command (cyl);

                //! wait for the results phase IRQ
                flpydsk_wait_irq ();
                flpydsk_check_int (&st0,&cyl0);

                //! found the cylinder?
                if ( cyl0 == cyl)
                        return 0;
        }

        return -1;
}
```

Invalid Commands

If an invalid command is sent to the FDC, the FDC ignores it and goes into standy mode.

Resetting the FDC

Disabling the Controller

If the DOR RESET line is low, the controller will be in a disabled state. In other words, just write 0 to the DOR register to disable the controller:

```
void flpydsk_disable_controller () {

        flpydsk_write_dor (0);
}
```

Enabling the Controller

To enable the controller, set the RESET line high in DOR. Also, because we want the FDC to operate in DMA mode, you must also set that bit in DOR:

```
void flpydsk_enable_controller () {

        flpydsk_write_dor ( FLPYDSK_DOR_MASK_RESET | FLPYDSK_DOR_MASK_DMA);
}
```

When the controller is enabled after being disabled, it will issue an interrupt. During this time, you must reinitialize the controller and drive configuation.

Initializing the FDC

During a controller reset, you need to reinitialize the controller. After resetting the controller, it will fire IRQ 6. After it has been fired, you must send a SENSE_INTERRUPT command to all drives connected to the FDC (by calling **flpydsk_check_int** 4 times.)

Afterwords its time to reconfigure the controller. Remember that the **CCR (Configuation Control Register)** only has 2 bits for the data rate. By setting both to 0, we set the data rate to 500 Kbps, which is a nice default value.

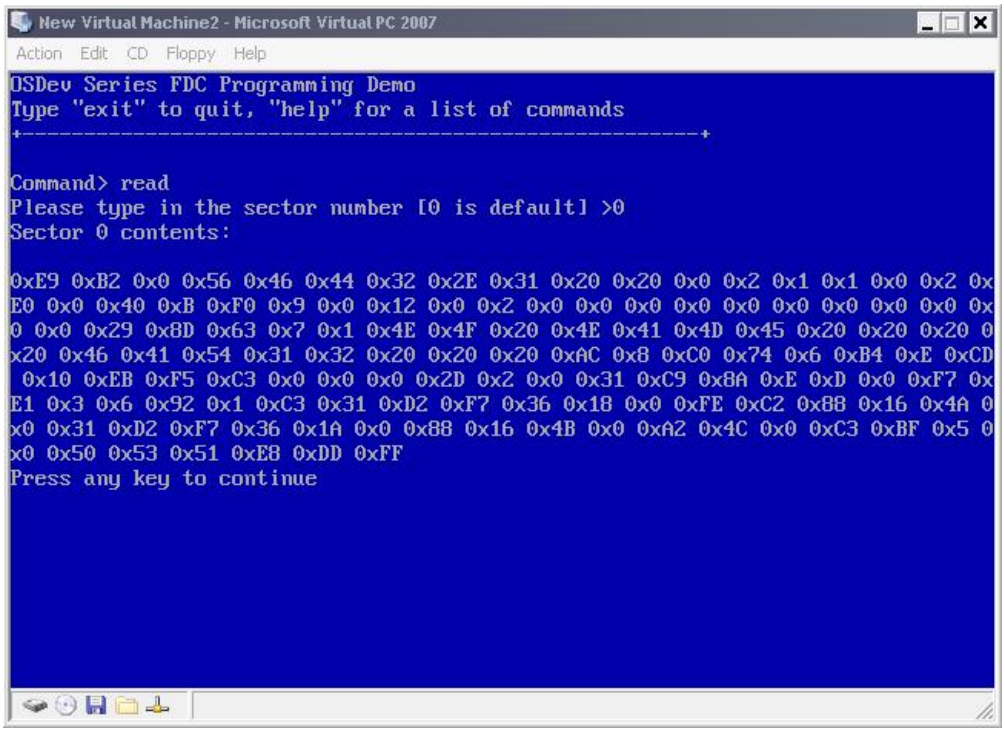
Then we call flpydsk_drive_data which sends a **Fix Drive Data / Specify** command to the controller to set the drives mechanical information, including: Step rate, head load and unload time, and if it supports DMA mode or not.

Then we recalibrate the drive so it is on cylinder 0.

```
void flpydsk_reset () {

        uint32_t st0, cyl;

        //! reset the controller
        flpydsk_disable_controller ();
        flpydsk_enable_controller ();
        flpydsk_wait_irq ();

        //! send CHECK_INT/SENSE INTERRUPT command to all drives
        for (int i=0; i<4; i++)
                flpydsk_check_int (&st0,&cyl);

        //! transfer speed 500kb/s
        flpydsk_write_ccr (0);

        //! pass mechanical drive info. steprate=3ms, unload time=240ms, load time=16ms
        flpydsk_drive_data (3,16,240,true);

        //! calibrate the disk
        flpydsk_calibrate ( _CurrentDrive );
}
```

After a reset, the drive is ready to be used by us.

Demo



FDC Demo in action
Demo Download

Note: There is a known bug in the demo that causes VPC to only read the first sector read. This will be resolved as soon as possible. No known issues when running in Bochs.

Updates and Changes

String to int convertion - stdio.h/stdio.cpp

In order to make this demo more interactive, I included three functions in the standard library that are used for converting strings into integers. This includes **strtol**, **strtoul** and **atoi**. The demo uses **atoi** to convert a string entered from the user into a useable integer.

Installing the floppy driver - flpydsk.cpp

The floppy driver comes with a nice install routine that allows the demo to easily set it up. All it does is install our interrupt handler using our HAL's setvect () routine, initializes the DMA for transfers, and resets the controller so it is ready for use.

```
void flpydsk_install (int irq) {

	//! install irq handler
	setvect (irq, i86_flpy_irq);

	//! initialize the DMA for FDC
	flpydsk_initialize_dma ();

	//! reset the fdc
	flpydsk_reset ();

	//! set drive information
	flpydsk_drive_data (13, 1, 0xf, true);
}
```

The demo calls this function during initialization to set up the driver before attempting to read from it.

Reading any sector - LBA and CHS - flpydsk.cpp

The driver hides the details of CHS behind two nice functions. Knowing that the drive works in CHS (Cylinder/Head/Sector) and does not know anything about LBA (Linear Block Addressing), we should provide a routine to convert between these two. This way we can just pass in a sector number to read or write to/from without worry of what physical CHS it is at.

Remember the formula to convert LBA to CHS? Lets apply it here:

```
void flpydsk_lba_to_chs (int lba,int *head,int *track,int *sector) {

   *head = ( lba % ( FLPY_SECTORS_PER_TRACK * 2 ) ) / ( FLPY_SECTORS_PER_TRACK );
   *track = lba / ( FLPY_SECTORS_PER_TRACK * 2 );
   *sector = lba % FLPY_SECTORS_PER_TRACK + 1;
}
```

FLPY_SECTORS_PER_TRACK is 18. Great! So now we can just call this function to convert any linear sector number into a CHS location! Cool, huh?

Because we are wanting to be able to read any sector from disk, we can provide a routine for just that. And because we already have **flpydsk_read_sector_imp**, which containes the code to send the read command to the controller, this routine is very simple.

```
uint8_t* flpydsk_read_sector (int sectorLBA) {

        if (_CurrentDrive >= 4)
                return 0;

        //! convert LBA sector to CHS
        int head=0, track=0, sector=1;
        flpydsk_lba_to_chs (sectorLBA, &head, &track, §or);

        //! turn motor on and seek to track
        flpydsk_control_motor (true);
        if (flpydsk_seek (track, head) != 0)
                return 0;

        //! read sector and turn motor off
        flpydsk_read_sector_imp (head, track, sector);
        flpydsk_control_motor (false);

        //! warning: this is a bit hackish
        return (uint8_t*) DMA_BUFFER;
}
```

Whenever the demo wants to read a sector, it calls this routine. This routine converts the sector into a physical location on disk (CHS). It turns the motor on and seeks to the cylinder that this sector is on. After words, it calls **flpydsk_read_sector_imp** to perform the magic of actually reading the sector and turns the motor off afterwords.

After the flpydsk_read_sector_imp call, the data for the sector should be in the DMA buffer. We return a pointer to this buffer, which now containes the sector data just read. Cool, huh?

This is the magical routine that ties everything together :)

New Read Command - main.cpp

This demo builds on the last demo. Because of this, it keeps the command line interface (CLI) that was built in the previous demo. This also makes this demo the most complex demo yet.

I have added a new command to our list of commands in the CLI - **read** - that allows us to read any sector off disk. It uses out floppy driver built in this tutorial to do it.

The command is inside of a function and is executed in the demo by typing **read**. It dumps the 512 bytes into 4 128-byte blocks for readability. After each block, you will be prompted to press a key to continue with the next chunk. It uses the new **atoi** function to convert the sector number entered (which is an LBA sector number) into an int, and reads it in. This, dear readers, is the function that makes the magic happen:

```
void cmd_read_sect () {

        uint32_t sectornum = 0;
        char sectornumbuf [4];
        uint8_t* sector = 0;

        DebugPrintf ("\n\rPlease type in the sector number [0 is default] >");
        get_cmd (sectornumbuf, 3);
        sectornum = atoi (sectornumbuf);

        DebugPrintf ("\n\rSector %i contents:\n\n\r", sectornum);

        //! read sector from disk
        sector = flpydsk_read_sector ( sectornum );

        //! display sector
        if (sector!=0) {

                int i = 0;
                for ( int c = 0; c < 4; c++ ) {

                        for (int j = 0; j < 128; j++)
                                DebugPrintf ("0x%x ", sector[ i + j ]);
                        i += 128;

                        DebugPrintf("\n\rPress any key to continue\n\r");
                        getch ();
                }
        }
        else
                DebugPrintf ("\n\r*** Error reading sector from disk");

        DebugPrintf ("\n\rDone.");
}
```

Conclusion

Yeesh, this is a long tutorial. I might be making some changes to help improve it and make it better and more complete. :)

In the next tutorial, we will be looking at the DMA. We will create an interface for programming the DMA and better use it in the FDC driver. After all of this...I suppose its filesystems again (ugh). Dont worry - After that it is Multitasking!

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

 Chapter 19

Home

Operating Systems Development Series

Operating Systems Development - 8237A ISA DMAC

by Mike, 2009

This series is intended to demonstrate and teach operating system development from the ground up.

Note: From here on out, demo names will follow the format Demo00, where 00 is the chapter name. This is to help the current issue with demo names and chapter names not being related to each other and to help make it easier for readers to know what chapter a particular demo refers to. Older chapters will be updated with this setup. Once all chapters have been updated this comment will be removed.

Also note: The Virtual PC bug has been fixed in this chapter but not the previous chapter yet. While the previous chapters demo does not work well in Virtual PC, this chapters demo seems to work fine with a minor bug fix and update in the DMA code. This demo has been tested and works on Virtual PC and Bochs. Once the previous chapter has been updated with this fix, this comment will be removed.

Introduction

Welcome! :)

In this chapter, we will take a close look at the **Direct Memory Access Controller (DMAC)**. The DMAC provides us a way to transfer blocks of data from a device directly into memory without the software doing it. This allows for a very fast way of transferring data as it is the hardware doing it - not the software.

Here is on the list for today:

- DMA History
- DMA Hardware
- DMA Ports
- DMA Registers
- DMA Commands

Abstract

Direct Memory Access (DMA) is a feature in all modern computers that allow devices to be able to move large blocks of data without any interaction with the processor. This can be useful, as you may have already seen from the floppy programming chapter. While the device transfers the block of data, the processor is free to continue running the software without worry about the data being transferred into memory, or to another device. The basic idea is that we can schedule the DMA device to perform the task on its own. Cool, huh?

Different buses and architecture designs have different methods of performing direct memory access. While our focus at this time will be the **ISA Direct Memory Access Controller (DMAC)** I decided to address the other methods as well for completeness.

ISA

The **Industry Standard Architecture (ISA)** provides a centric location for DMA requests through a controller based off of the **Intel 8237 Microcontroller**. In the ATX motherboard designs, there was only a single controller. Do to the limitations of a controller only supporting 8 devices, however, in AT and newer architectures there are **two** controllers. They are slaved together, in a similar way the **Programmable Interrupt Controllers (PIC)** are slaved together. **Both controllers always run at 4MHz.**

Because of their performance and limited number of devices, newer devices tend to us PIO or UDMA instead. DMA is still supported in ISA for legacy devices however.

All of these devices are connected to **Channels** on the controller. Along with these channels, each channel has a **DACK (DMA Acknowledge)** line and a **DRQ (DMA Request)**.

Here are the standard assignments on both **Direct Memory Access Controllers (DMAC)**.

- XT:
  - **Channel 0:** Used by system, not available (DRAM Refresh, obsolete)
  - **Channel 1:** Available, no standard DMA assignment
  - **Channel 2:** Floppy Disk Controller
  - **Channel 3:** Hard Disk Controller (PIO or UDMA recommended instead)
- AT only:
  - **Channel 4:** Cascaded to XT controller - Slave DMA controller input into Master
  - **Channel 5:** Available, no standard DMA assignment (16 bit)
  - **Channel 6:** Available, no standard DMA assignment (16 bit)

- **Channel 7:** Available, no standard DMA assignment (16 bit)

To start a transfer, the software sets the channels address and count registers to the location in physical memory where the transfer is to be completed, and the size of the transfer. Afterwords it sets to either read or write from that block of memory and then sets the controller on its way to complete the transfer. After the transfer completes, the device that started the transfer issues an **Interrupt Request (IRQ)** to be caught by the system software for further processing. **This is important!** These are the steps that we will need to perform when using the DMA to start transfers.

PCI

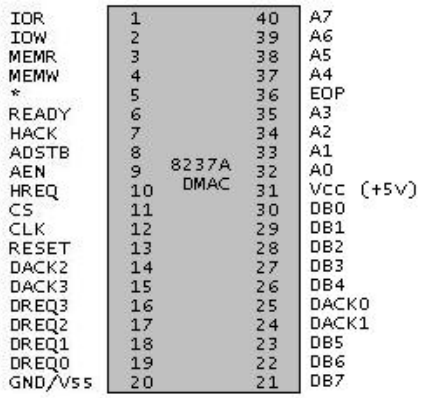
PCI devices do not share the same DMA controller, nor have a central DMA controller. Instead, a PCI device on the **PCI Local Bus** requests to be the **Bus Master** (taking control of the bus) from the **PCI Bus Controller**. Afterwords, a request to read or write to or from physical memory is passed to the northbridge which will convert the request into memory operations and send the operations to the **Memory Controller**.

PCI transfers are limited to 4GB physical memory. However if the device and the PCI bridge impliments **Double Address Cycle (DAC)** or similar technology, it will allow the PCI controller to initiate requests for reading and writing beyond 4GB physical memory.

ISA DMA Hardware

Direct Memory Access Controller (DMAC)

The **Industry Standard Architecture (ISA)** uses a controller based off of the original **Intel 8237 DMA** chip. Most newer DMACs provide more features, but are almost entirely backward compatable with the 8237 microcontroller. While new PCs have a more advanced form of the DMAC, it is always nice to look at the device that started it all, so here it is, the original 8237A controller pin diagram when distributed in a **Dual Inline Package (DIP)**:



Thats it - the controller that we will be programming in this chapter. THere are alot of pins, but its not too complex. Lets look at them, and focus on the important ones.

- **Pin 1 (IOR)** I/O Read
- **Pin 2 (IOW)** I/O Write
- **Pin 3 (MEMR)** Memory Read
- **Pin 4 (MEMW)** Memory Write
- **Pin 5**
- **Pin 6 (READY)**
- **Pin 7 (HACK)** Hold Acknowledge
- **Pin 8 (ADSTB)** Address Strobe
- **Pin 9 (AEN)** Address Enable
- **Pin 10 (HREQ)** Hold Request
- **Pin 11 (CS)** Chip Select
- **Pin 12 (CLK)** Clock
- **Pin 13 (RESET)** Reset
- **Pins 14-15 (DACK)** DMA Ackowledge
- **Pins 16-19 (DREQ0-DREQ3)** DMA Request
- **Pin 20 (GND/Vss)** Ground
- **Pins 21-23 (DB0-DB3)** Data Bus
- **Pins 24-25 (DACK)** DMA Ackowledge
- **Pins 26-30 (DB4-DB7)** Data Bus
- **Pin 31 (Vcc)** +5 volt power
- **Pins 32-35 (A0-A3)** Address Lines
- **Pin 36 (EOP)** End Of Process
- **Pins 37-40 (A4-A7)** Address Lines

This one is not too bad. We have **Pin 20** for the ground, and **pin 31** for the power source. **Pin 12 (CLK)** is another common one we see on all controllers. It connects to the processors CLK pin for input clock signals: controlling the timing of operations within the controller. The **CS (Chip Select)** pin is another common one we see on almost all controllers. Its used to select the controller as an I/O device on the data bus. **RESET** resets the controllers internal registers (Status,

Request, Temporary, Command), clears the internal flip-flop and sets the mask register. Nothing much new so far, huh? We have the gerneric address lines, **A0-A7**, which connect to the systems address bus. During inputs, the CPU is only able to write data to **A0-A3** to select registers to read from. All pins are used for outputs (to a physical memory address) but are only activated during a DMA request. Last but not least is the generic **D0-D7** pins that connect to the systems data bus.

Now for the more interesting pins. So far we have seen that the DMAC connects to the systems address and data bus. Alot of our readers probably are not too surprised about that. As you can probably guess, however, the DMAC needs direct attention from the CPU. Because of this, there are some lines that connect to the CPU so that the DMAC can communicate with the CPU and vice-versa. This is done with the **HACK** and **HREQ** pins. **HACK (Hold Acknowledge)** is held high when the CPU has given the DMAC full control of the system bus. This allows the DMAC to know when it is safe to transmit data to the memory controller. After all, we cannot have both the DMAC and processor trying to use the same system bus at the same time, can we?

This creates an important note: **The DMAC transmits data directly into physical memory only when the system bus is not currently being used by the processor.** The DMAC will need the system bus to transmit data to the memory controller for memory translation and reading/writing to physical memory.

Okay, so the CPU has a way to tell the DMAC the system bus can be taken over. Great, but how does the DMAC tell the CPU it needs the system bus in the first place? Thats what the **HREQ (Hold Request)** line is for. When an **DMA Request (DRQ)** is triggered by a device connected to the DMA (such as a floppy controller), and that "channel" is currently not disabled, the controller puts HREQ to high on the next clock cycle to notify the CPU that it needs control of the system bus in order to complete the request.

The lines **DR0-DR3 (DMA Request Lines)** are used by devices to notify the DMA of a request. For example, the **Floppy Drive Controller (FDC)** is usually connected to use **DR2** ("Channel 2"). So, when we have enabled that channel and programmed the FDC to use the DMAC, when a read or write command is sent to the FDC, the FDC will activate the **RQ2** line notifying the DMAC that it requires attention. From here everything is done through the DMAC to read or write depending on the mode that we programmed that channel to be in.

This creates another important point. Knowing that there are only 4 DRQ lines: **Only 4 devices can be connected to a single DMAC.** This is pretty limited, huh? In the i86 architecture, the problem has been somewhat solved by attaching two DMACs together. We will look at that shortley.

Everything is looking good so far! Knowing that our software instructs the CPU to program the DMAC, how does the CPU tell the DMAC that we are in needing to read or write from a register? Thats the **IOR (I/O Read)** and **IOW (I/O Write)** pins. In a similar way, the DMAC tells the memory controller that it will be reading or writing by activating the memory read or write control lines by outputting through **MEMR (Memory Read)** or **MEMW (Memory Write)**. **EOP (End Of Process)** is used to signal a device when the request has been completed. A request is completed when that channels **Terminal Count (TC)** has been reached. This is a programmable counter value. **AEN (Address Enable)** is used to signal the controller to load its internal 8 bit address latch register into the systems address bus. **ADSTB (Address Strobe)** is used to strobe the upper address byte into an external latch register.

Alot of things, huh? The exact details of the operations that the controller takes depends on the mode that it is in and the transfer type. It uses the same basic steps though: A device notifies the DMAC, the DMAC notifies the CPU for control over the system bus. The DMAC waits for control. When it gets it, it loads the channels address register into its internal latch register. From there, it will either set MEMR, MEMW, and read or write from memory as needed. Wait, what? I am sure you can see how it can read from memory, but if it writes it, where does it go?

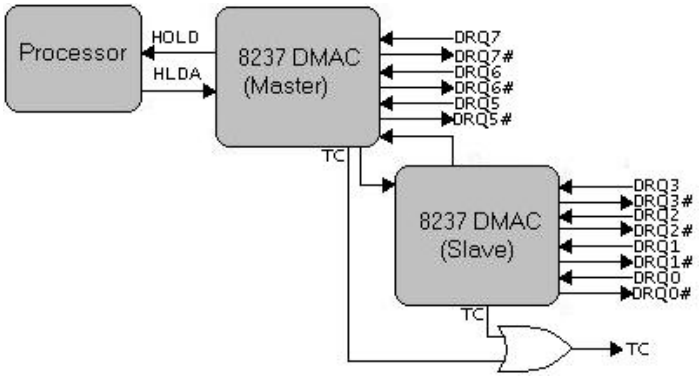
Look back at the FDC chapter again. Notice how it also has pins **D0-D7** that connect to the same data bus that the DMAC, CPU, Memory Controller, and other devices see? So, when writing from memory all it needs to do is activate its MEMR line, upload the address to the address bus, the memory controller translates and places the data on the data bus. Because the FDC is waiting for a write request, it will grab the data read, and write to disk set up by the write command that was sent to the FDC. When reading from disk, its basically the same way buth the DMAC will activate the MEMW line instead. The memory controller grabs the data to be written from the data bus sent by the FDC. When all is good to go, the DMAC releases the **HREQ** line on the processor which gives the CPU full control of the bus again.

Its important to note that the processor cannot wait for the DMAC to finish. The CPU will bring the **HACK** line to low when it needs access to the system bus again. During these periods, the DMAC will have no choice but to wait until the line is high again to continue its process.

And there you go, dear readers! As you recall, in the i86 architecture things are a little different. i86 added another DMAC to the mix to bring the number of channels that can be used to 8. Well, sort of. Lets take a look!

DMAC in x86

Remember that newer PCs have two DMACs? Both DMACs are connected in a similar way that the two PICs are connected together...Only backwards. Huh!? I know, I know. :) Lets take a look:



The DMAC uses the **HOLD** and **HLDA (Hold Acknowledge)** pins on the processor when taking control of the ISA bus. The DMAC signals the processor through HOLD, and the processor acknowledges this request through HDLA. Also note how the second (slave) containes DRQ's 0-3 while the primary DMAC has DRQ's 5-7. A DRQ is a **DMA Request**. These lines connect the DMAC to different devices in the system that use it. Whenever a device requests the DMACs attention, it raises the line to high to signal the DMAC. Look at the image again and you might see something interesting: **Where is DRQ4?**

DRQ4 does exist on both of the devices, but are what connects each DMAC. They are shown in the image (not labeled). Because DRQ lines are used to signal the DMAC, this allows the primary and secondary DMACs to signal each other to raise correct DRQ lines. **This means when programming the DMACs, we have to remember that DRQ4 is used to connect the primary and slave controllers. Because of this we cannot use it.** Looking back at the image above, we also see an **OR Gate** that will output true if either the primary or slave DMACs are complete (they raise their **TC (Terminal Count)** line). The TC line will raise when the transfer request that was sent to the DMAC has been completed.

Okay, so lets put everything important that we need to remember here for reference.

- The DMA always works in physical memory, never virtual memory
- Only 8 devices can be connected to use the DMACs on the i86 architecture.
- DRQ4 (Channel 4) is used to connect the primary and secondary DMACs and cannot be used.

You may also see something interesting about how these are configured. We have the slave DMAC which is the **first** DMAC that connects to the Master DMAC, **not** the other way around. This will explain why the slave DMAC is responsible for channels 0-3 (and technically 4, which is used to connect to the primary DMAC) and the primary DMAC is responsible for the channels 5-7. Kind of weird, huh? In this way, its somewhat different then the way the two PICs work together. It is also important to note, do to the way these controllers are connected together, **the master DMAC acts like a 16 bit DMAC while the slave DMAC acts like an 8 bit DMAC**. Because of this:

- **The first DMAC is the slave (8 bit), the second is Master (16 bit)**

ISA DMA Interface

Port Mapping

Because there are two DMA controllers, there are two sets of ports.

Generic Registers

ISA DMAC Ports		
DMAC 0 Port (Slave)	DMAC 1 Port (Master)	Descripton
0x08	0xD0	Status Register (Read)
0x08	0xD0	Command Register (Write)
0x09	0xD2	Request Register (Write)
0x0A	0xD4	Single Mask Register (Write)
0x0B	0xD6	Mode Register (Write)
0x0C	0xD8	Clear Byte Pointer Flip-Flop (Write)
0x0D	0xDA	Intermediate Register (Read)
0x0D	0xDA	Master Clear (Write)
0x0E	0xDC	Clear Mask Register (Write)
0x0F	0xDE	Write Mask Register (Write)

These registers will be described in more detail in the next section. These registers are used when interacting with both DMACs. They can be read or written to through port mapped I/O. That is, using standard i86 **in** and **out** instructions.

It is very important to remember that DMACs are backwards. DMAC 0 is the **slave** while DMAC 1 is the **master**. Also note how the port ranges are different. Remember that the slave is 8 bit, while the master is 16 bit?

To help readability, lets abstract these ugly numbers in an enumeration:

```
enum DMA0_IO {

        DMA0_STATUS_REG = 0x08,
        DMA0_COMMAND_REG = 0x08,
        DMA0_REQUEST_REG = 0x09,
        DMA0_CHANMASK_REG = 0x0a,
        DMA0_MODE_REG = 0x0b,
        DMA0_CLEARBYTE_FLIPFLOP_REG = 0x0c,
        DMA0_TEMP_REG = 0x0d,
        DMA0_MASTER_CLEAR_REG = 0x0d,
        DMA0_CLEAR_MASK_REG = 0x0e,
        DMA0_MASK_REG = 0x0f
};
```

Notice that these values match up with the table above. Now for DMAC 2...

```
enum DMA1_IO {

        DMA1_STATUS_REG = 0xd0,
        DMA1_COMMAND_REG = 0xd0,
        DMA1_REQUEST_REG = 0xd2,
        DMA1_CHANMASK_REG = 0xd4,
        DMA1_MODE_REG = 0xd6,
        DMA1_CLEARBYTE_FLIPFLOP_REG = 0xd8,
        DMA1_INTER_REG = 0xda,
        DMA1_UNMASK_ALL_REG = 0xdc,
        DMA1_MASK_REG = 0xde
};
```

Now on with the registers!

Channel Registers

Along with the above registers, the i86 makes available the following registers that allow us to control the address and counters of each channel:

ISA DMAC Channel Ports		
DMAC 0 Port (Slave)	DMAC 1 Port (Master)	Descripton
0x0	0xC0	Channel 0 Address/Channel 4 Address
0x1	0xC2	Channel 0 Counter/Channel 4 Counter
0x2	0xC4	Channel 1 Address/Channel 5 Address
0x3	0xC6	Channel 1 Counter/Channel 5 Counter
0x4	0xC8	Channel 2 Address/Channel 6 Address
0x5	0xCA	Channel 2 Counter/Channel 6 Counter
0x6	0xCC	Channel 3 Address/Channel 7 Address
0x7	0xCE	Channel 3 Counter/Channel 7 Counter

Look at the table above again. Channel 0 Address on the master DMAC is at .. what? port 0! This is a historical moment in this series as we have found i/o port 0. :)

Again remeber that the **primary DMAC is DMAC 1** while the slave DMAC is DMAC 0. Also remember how the master DMAC was 16 bits while the slave DMAC was 8? This an important characteristic, specifically here as **this means you can read or write 8 bit values to the slave DMAC, but 16 bit values to the master DMAC.**

Anyways, before we get into the details about these registers lets first hide them. Looking at the enumerations below, you will see nothing fancy going on - they match the tables above. Remember that these registers are all accessed through port mapped i/o. In other words, you can read or write them using **in** and **out** x86 machine instructions.

```
enum DMA0_CHANNEL_IO {

        DMA0_CHAN0_ADDR_REG = 0, //! Thats right, i/o port 0
        DMA0_CHAN0_COUNT_REG = 1,
        DMA0_CHAN1_ADDR_REG = 2,
        DMA0_CHAN1_COUNT_REG = 3,
        DMA0_CHAN2_ADDR_REG = 4,
        DMA0_CHAN2_COUNT_REG = 5,
        DMA0_CHAN3_ADDR_REG = 6,
        DMA0_CHAN3_COUNT_REG = 7,
};
```

...and now for DMAC 2...

```
enum DMA1_CHANNEL_IO {

        DMA1_CHAN4_ADDR_REG = 0xc0,
        DMA1_CHAN4_COUNT_REG = 0xc2,
        DMA1_CHAN5_ADDR_REG = 0xc4,
        DMA1_CHAN5_COUNT_REG = 0xc6,
        DMA1_CHAN6_ADDR_REG = 0xc8,
        DMA1_CHAN6_COUNT_REG = 0xca,
        DMA1_CHAN7_ADDR_REG = 0xcc,
        DMA1_CHAN7_COUNT_REG = 0xce,
};
```

The basic purpose of these registers is to provide a way for us to tell the DMAC how to initiate the channels. Each channel has a base address and a counter. The base address is the location in memory to start reading or writing, and the counter tells the DMAC how much to transfer on that channel. **It is important to note that these are always physical addresses, not virtual!**

Lets have an example. To set the base address that a channel will use, all we need to do is write to the correct i/o port shown in the above table. Assuming **DMA0_CHAN0_ADDR_REG** is 0 all the way to **DMA1_CHAN7_ADDR_REG** being the last value in the table (0xde), this becomes easy. **Remember that all example code is in the demo at the end of this chapter.**

```
void dma_set_address(uint8_t channel, uint8_t low, uint8_t high) {

        if ( channel > 8 )
                return;

        unsigned short port = 0;
        switch ( channel ) {

                case 0: {port = DMA0_CHAN0_ADDR_REG; break;}
                case 1: {port = DMA0_CHAN1_ADDR_REG; break;}
                case 2: {port = DMA0_CHAN2_ADDR_REG; break;}
                case 3: {port = DMA0_CHAN3_ADDR_REG; break;}
                case 4: {port = DMA1_CHAN4_ADDR_REG; break;}
                case 5: {port = DMA1_CHAN5_ADDR_REG; break;}
                case 6: {port = DMA1_CHAN6_ADDR_REG; break;}
                case 7: {port = DMA1_CHAN7_ADDR_REG; break;}
        }

        outportb(port, low);
        outportb(port, high);
}
```

In a very similar way, we can write a routine to set the count register of that specific channel.

```
void dma_set_count(uint8_t channel, uint8_t low, uint8_t high) {

        if ( channel > 8 )
                return;

        unsigned short port = 0;
        switch ( channel ) {

                case 0: {port = DMA0_CHAN0_COUNT_REG; break;}
                case 1: {port = DMA0_CHAN1_COUNT_REG; break;}
                case 2: {port = DMA0_CHAN2_COUNT_REG; break;}
                case 3: {port = DMA0_CHAN3_COUNT_REG; break;}
                case 4: {port = DMA1_CHAN4_COUNT_REG; break;}
                case 5: {port = DMA1_CHAN5_COUNT_REG; break;}
                case 6: {port = DMA1_CHAN6_COUNT_REG; break;}
                case 7: {port = DMA1_CHAN7_COUNT_REG; break;}
        }

        outportb(port, low);
        outportb(port, high);
}
```

It is very important to note that these registers are 16 bits. This means that the DMAC can only transfer 64k at most at a time from a single channel.

It is also very important to note that these are physical addresses! If the system software has enabled paging, it must map the location that the channel will use into the same virtual address by **idenitity mapping** the region of memory that will be used.

So, to recap: Knowing that there are 8 channels, after enabling the device on that channel to use the DMAC, we can initiate a read or write transfer to the DMAC by giving the channel information (the memory location, and weather to read or write to or from it.) by writing to one of the channel registers. You might be asking where this data comes from. Or, if reading, where does the data go? Thats up to the device that is controlling that channel. For example, in a floppy drive, after we send a read command to the Floppy Drive Controller (FDC), the FDC will notify the DMAC to initiate the transfer. The DMAC will get the base physical address, channel operation (read or write, hopefully read in this case ;)), and size of the buffer and the rest writes itself: The FDC will continue to transfer data to the DMAC which in turn will place it in the buffer that is pointed by the address stored in that channel. **We set the location of the buffer and the size of it here, by writing to that channels address and count registers.**

Wait, wait, wait. Remember that the DMAC can only transfer 64K at a time? Its worse then that. Knowing the base address of each channel also has the same limitation, this also means the DMACs are limited to accessing 64K of RAM! This is a bad limitation, dont you think? A solution to this is the external page registers. Lets look closer!

Extended Page Address Registers

The Page Registers are used to set what **page** the memory location that the channel is set to resides. Each page register is 8 bits, and there is a page register for each channel (well, actually more then that but that is not important.) If we take these 8 bits and append them to the base address of the channel (making 0xFFFFFF for a channel base address), we effectively have 8 more bits so we can access up to 16MB of memory. This is how these page registers work.

These page registers only store the upper 8 bits of that channels transfer address. This is an important characteristic as it means the values in these page registers are always a multiple of 64k.

Sure enough, things get a little messy here. The original PCs having one DMAC used different i/o ports then AT/EISA/MCA and newer computers. And because the newer computers have two DMACs, additional registers were added and extended with more bits. The original PC page registers only added 4 bits (A16-A19). The newer computers, on the other hand, added 8 bits (A16-A23) to the base channel address.

ISA DMAC Extended Page Address Registers	
Port	Descripton
0x80	Channel 0 (Original PC) / Extra / Diagnostic port
0x81	Channel 1 (Original PC) / Channel 2 (AT)
0x82	Channel 2 (Original PC) / Channel 3 (AT)
0x83	Channel 3 (Original PC) / Channel 1 (AT)
0x84	Extra
0x85	Extra
0x86	Extra
0x87	Channel 0 (AT)
0x88	Extra
0x89	Channel 6 (AT)
0x8A	Channel 7 (AT)
0x8B	Channel 5 (AT)
0x8C	Extra
0x8D	Extra
0x8E	Extra
0x8F	Channel 4 (AT) / Memory refresh / Slave Connect

Okay, lets stop for a moment. *grabs a cup of coffie* Okay, all that we need to concern ourself with in the above table is the ports listed for AT. This means that all channel external page registers add an additional 8 bits to the base address of the channel that we stored when setting up the channel (see previous section.) Knowing that the page registers are only the top 8 bits, this means that the values in these registers must be a multiple of 64k. For example, when programming the floppy controller, we know that the floppy uses DMA channel 2. Lets say we want to store a buffer somewhere lower then 64k, we can just set that channels address to point somewhere right? Well, kind of. We will also need to set its page register because it is used to determin the top 8 bits of that address. So, to set it:

- If set to 0: Page 0, nothing added to the address
- If set to 1: Page 1, 64k added to the address
- If set to 2: Page 2, 128K added to the address
- If set to 255, Page 255 = 255*64K=0xFF0000, all top 8 bits set, 16, 320K or about 16 MB added to the address

Notice how changing the page in the page table changes the address where the DMA is to read or write to. This allows the DMAC to effectively access up to 16MB of memory. Cool, huh? Still a little limiting, but alot better then being limited to 64K dont you think?

Like the other registers, lets hide those ugly magic numbers:

```
enum DMA0_PAGE_REG {

        DMA_PAGE_EXTRA0 = 0x80, //! Also diagnostics port
        DMA_PAGE_CHAN2_ADDRBYTE2 = 0x81,
        DMA_PAGE_CHAN3_ADDRBYTE2 = 0x82,
        DMA_PAGE_CHAN1_ADDRBYTE2 = 0x83,
        DMA_PAGE_EXTRA1 = 0x84,
        DMA_PAGE_EXTRA2 = 0x85,
        DMA_PAGE_EXTRA3 = 0x86,
        DMA_PAGE_CHAN6_ADDRBYTE2 = 0x87,
        DMA_PAGE_CHAN7_ADDRBYTE2 = 0x88,
        DMA_PAGE_CHAN5_ADDRBYTE2 = 0x89,
        DMA_PAGE_EXTRA4 = 0x8c,
        DMA_PAGE_EXTRA5 = 0x8d,
        DMA_PAGE_EXTRA6 = 0x8e,
        DMA_PAGE_DRAM_REFRESH = 0x8f //!no longer used in new PCs
};
```

To set one of these registers, all we need to do is determin what register is being written to (based on what channel is passed to it) and write the value to it:

```
void dma_set_external_page_register (uint8_t reg, uint8_t val) {

        if (reg > 14)
                return;

        unsigned short port = 0;
        switch ( reg ) {

                case 1: {port = DMA_PAGE_CHAN1_ADDRBYTE2; break;}
                case 2: {port = DMA_PAGE_CHAN2_ADDRBYTE2; break;}
                case 3: {port = DMA_PAGE_CHAN3_ADDRBYTE2; break;}
                case 4: {return;}//! nothing should ever write to register 4
                case 5: {port = DMA_PAGE_CHAN5_ADDRBYTE2; break;}
                case 6: {port = DMA_PAGE_CHAN6_ADDRBYTE2; break;}
                case 7: {port = DMA_PAGE_CHAN7_ADDRBYTE2; break;}
        }

        outportb(port, val);
}
```

It is important to note that case 4 is commented. Remember that channe 4 is used to cascade with the master DMAC? Because if this, nothing can use it. Each of the above cases represent a channel that we are setting the page to. So, a call like **dma_set_external_page_register (2, 0x1000);** will allow us to set 0x1000 to the channel 2 page register. Cool?

Registers

In addition to the registers shwn above, the controller makes the following registers available as well.

Command Register

This register is used to control the DMAC. It has the following format:

- **Bit 0: MMT** Memory to Memory Transfer
  - **0:** Disable
  - **1:** Enable
- **Bit 1: ADHE** Channel 0 Address Hold
  - **0:** Disable
  - **1:** Enable
- **Bit 2: COND** Controller Enable
  - **0:** Disable
  - **1:** Enable
- **Bit 3: COMP** Timing
  - **0:** Normal
  - **1:** Compressed
- **Bit 4: PRIO** Priority
  - **0:** Fixed Priority
  - **1:** Normal Priority
- **Bit 5: EXTW** Write Selection
  - **0:** Late Write Selection
  - **1:** Extended Write Selection
- **Bit 6: DROP** DMA Request (DREQ)

    - **0:** DREQ sense active high
    - **1:** DREQ sense active low
- **Bit 7: DACKP** DMA Acknowledge (DACK)
    - **0:** DACK sense active low
    - **1:** DACK sense active high

Most of these bits will not work on the i86 architecture. The only bit that does work is bit 2, which can be used to enable or disable the controller. I know, I know, you would think direct memory to memory transfers would be useful too. Using other bits may either not do anything, or provide unpredictable results.

For completeness, these are included in the dma.h header file in the demo at the end of this chapter. Here they are as bit masks.

```
enum DMA_CMD_REG_MASK {

        DMA_CMD_MASK_MEMTOMEM = 1,
        DMA_CMD_MASK_CHAN0ADDRHOLD = 2,
        DMA_CMD_MASK_ENABLE = 4,
        DMA_CMD_MASK_TIMING = 8,
        DMA_CMD_MASK_PRIORITY = 0x10,
        DMA_CMD_MASK_WRITESEL = 0x20,
        DMA_CMD_MASK_DREQ = 0x40,
        DMA_CMD_MASK_DACK = 0x80
};
```

Mode Register (Write)

This mode sets the mode of the controller. It has the following format:

- **Bits 0-1: SEL0, SEL1** Channel Select
    - **00:** Channel 0
    - **01:** Channel 1
    - **10:** Channel 2
    - **11:** Channel 3
- **Bits 2-3: TRA0, TRA1** Transfer Type
    - **00:** Controller self test
    - **01:** Write Transfer
    - **10:** Read Transfer
    - **11:** Invalid
- **Bit 4: AUTO** Automatic reinitialize after transfer completes (Device must support!)
- **Bit 5: IDEC**
- **Bits 6-7: MOD0, MOD 1** Mode
    - **00:** Transfer on Demand
    - **01:** Single DMA Transfer
    - **10:** Block DMA Transfer
    - **11:** Cascade Mode

This register is important. In order for us to set up a channel and prepare it to read or write a block of memory, we must write to this register the operation mode. Before writing to this register however, it is always recommended to mask off (disable) the channel you would like to set the mode for before changing anything. This has to do with the problem of changing channel modes while it is currently in use, which can cause data corruption or other issues.

Before anything, the first thing I always like to do is to hide the ugly numbers behind meaningful names, so here they are. This is a little different though: These enums are a combination of masks and flags. The masks match the bit format of the above list. The flags are there just for simplicity: They will allot us to set or clear the needed bit in the above list allowing us to bitwise-OR options together. So, for example, we can combine the channel number and set the mode of the channel to read a single transfer with auto initialize by just doing: channel | DMA_MODE_READ_TRANSFER | DMA_MODE_MASK_AUTO | DMA_MODE_TRANSFER_SINGLE. Cool, huh?

Because the format is the same for both controllers, we only have one enum:

```
enum DMA_MODE_REG_MASK {

        DMA_MODE_MASK_SEL = 3,

        DMA_MODE_MASK_TRA = 0xc,
        DMA_MODE_SELF_TEST = 0,
        DMA_MODE_READ_TRANSFER =4,
        DMA_MODE_WRITE_TRANSFER = 8,

        DMA_MODE_MASK_AUTO = 0x10,
        DMA_MODE_MASK_IDEC = 0x20,

        DMA_MODE_MASK = 0xc0,
        DMA_MODE_TRANSFER_ON_DEMAND= 0,
```

```
        DMA_MODE_TRANSFER_SINGLE = 0x40,
        DMA_MODE_TRANSFER_BLOCK = 0x80,
        DMA_MODE_TRANSFER_CASCADE = 0xC0
};
```

Assuming **DMA0_MODE_REG** is 0x0b - the DMA 0 mode register, and **DMA1_MODE_REG** is 0xd6, the second DMA mode register, all we need to do to set the DMA mode for a specific channel is this:

```
void dma_set_mode (uint8_t channel, uint8_t mode) {

        int dma = (channel < 4) ? 0 : 1;
        int chan = (dma==0) ? channel : channel-4;

        dma_mask_channel (channel);
        outportb ( (channel < 4) ? (DMA0_MODE_REG) : DMA1_MODE_REG, chan | (mode) );
        dma_unmask_all ( dma );
}

//! prepares channel for read
void dma_set_read (uint8_t channel) {

        dma_set_mode (channel,
                DMA_MODE_READ_TRANSFER | DMA_MODE_TRANSFER_SINGLE | DMA_MODE_MASK_AUTO);
}

//! prepares channel for write
void dma_set_write (uint8_t channel) {

        dma_set_mode (channel,
                DMA_MODE_WRITE_TRANSFER | DMA_MODE_TRANSFER_SINGLE | DMA_MODE_MASK_AUTO);
}
```

This routine will allow us to set the mode on any channel. Cool, huh? For example, if we want to prepare the floppy drive to write, a **dma_set_mode (2, 0x5A)** will do it. (Remember that the floppy uses channel 2 on the primary DMAC?) and 0x56 = 01010110 binary. Comparing it with the list above, Mode=01 (Single transfer), AutoInit is set (Auto initialize after completion), transfer type=01 (Write), channel 2 (10).

The DMA_MODE_MASK_AUTO bit is a useful one. It allows us to initialize the DMAC once at the start (By resetting the controller and setting the channels buffer address and count) without needing to worry about it again. If this bit is not set, we will need to reinitialize the DMAC before every read or write operation.

Note: The AutoInit bit (DMA_MODE_MASK_AUTO) does not seem to be supported well in Virtual PC. Because of this, to help maintain portability with Virtual PC, we opted to go for reinitializing the DMAC every read or write operation rather then using AUTOINIT. Other emulators or machines may or may not support it.

Request Register (Write)

This register allows software to send to the DMAC directly. The first 2 bits are used to select the channel. For example, 00=channel 0, 01=channel 1, 10=channel 2, 11=channel 3. The third bit, if 0, resets the channel request bit. If 1, sets the request bit.

- **Bit 0-1:** Channel select 0
- **Bit 2:** 0=reset channel request bit, 1=set request bit

The Request Register is used for Memory-to-memory operations. Remembering form the command register, you cannot/should not enable memory-to-memory transactions in the i86 architecture. Because of this, this register is not important to us.

Channel Mask Register (Write)

This register allows you to be able to mask a single DMA channel. Bits 0 and 1 allow you to set the channel (00=channel 0, 01=channel 1, 10=channel 2, 11=channel 3). Bit 4 determins weather to mask or unmask the channel. If bit 4 is 0, it unmasks the channel. If it is 1, it will mask it. All other bits are unused.

- **Bit 0-1:** Channel select
- **Bit 2:** 0=unmasks channel, 1=masks channel

All other bits unused.

Mask Register (Write)

This register containes information on what channels are currently masked and unmasked. The top 4 bits in this 8 bit register are always unused. The low four bits are used to mask or unmask one of the four channels. For example, bit 0 is for channel 0, bit 1 is for channel 1, and so on. **Note: Masking channel 4 will also mask channels 4,5,6,7 due to cascading.**

- **Bit 0:** Channel select 0
- **Bit 1:** Channel select 1
- **Bit 2:** Channel select 2
- **Bit 3:** Channel select 3

All other bits unused.

For example, lets provide a routine to mask (disable) any channel, all we need to do is set the respective bit:

```
void dma_mask_channel(uint8_t channel){

        if (channel <= 4)
                outportb(DMA0_CHANMASK_REG, (1 << (channel-1)));
        else
                outportb(DMA1_CHANMASK_REG, (1 << (channel-5)));
}
```

In a similar way, to unmask a channel, just clear the bit:

```
void dma_unmask_channel (uint8_t channel) {

        if (channel <= 4)
                outportb(DMA0_CHANMASK_REG, channel);
        else
                outportb(DMA1_CHANMASK_REG, channel);
}
```

Both of these routines assumes that DMA0_CHANMASK_REG is 0x0a (The i/o port for the DMAC mask register) and DMA1_CHANMASK_REG is 0xD4 (The i/o port for the second DMAC mask register.)

Because you can set multiple channels at the same time, this register does allow the ability of masking or unmasking multiple channels at the same time.

Status Register

The status register has the following format:

- **Bit 0: TC0** Set if Channel 0 has reached **Transfer Complete (TC)**
- **Bit 1: TC1** Set if Channel 1 has reached **Transfer Complete (TC)**
- **Bit 2: TC2** Set if Channel 2 has reached **Transfer Complete (TC)**
- **Bit 3: TC3** Set if Channel 3 has reached **Transfer Complete (TC)**
- **Bit 4: REQ0** Set if Channel 0 is pending a **DMA Request (DRQ)**
- **Bit 5: REQ1** Set if Channel 1 is pending a **DMA Request (DRQ)**
- **Bit 6: REQ2** Set if Channel 2 is pending a **DMA Request (DRQ)**
- **Bit 7: REQ3** Set if Channel 3 is pending a **DMA Request (DRQ)**

This register is not very useful. In most cases, the device that is controlling the DMAC will send an IRQ when the transfer is complete, so there is no need to poll the register for information. The first 4 bits tell you if the transfer on that channel is complete. the last 4 bits tell you if the channel has pending DMA requests.

ISA DMA Commands

The controller provides special registers that allows software to be able to send commands to the controller. These commands do not at all require any specific bit format, and can be activated by a simple i/o operation.

The DMAC will recognize the command by the data on the address bus (lines A0-A3) and the status of its ORQ and IOW lines.

Please note that there is nothing special about these registers. These are also in the table of generic registers near the beginning of this chapter.

Clear Byte Pointer Flip-Flop

This is a special i/o address port that allows us to control the flip-flop between 16 bit transfers when working with the 8 bit DMAC (the primary DMAC.)

There are two ports for both DMACs:

ISA DMAC Flip-Flop Ports	
Port	Descripton
0x0C	DMAC 0 (16 bit) Slave (write)
0xD8	DMAC 1 (8 bit) Master (write)

For example, assuming **DMA0_CLEARBYTE_FLIPFLOP_REG** is 0x0c and **DMA1_CLEARBYTE_FLIPFLOP_REG** is 0xD8, the following routine will set or clear the flipflop:

```
void dma_reset_flipflop(int dma){

        if (dma < 2)
                return;

        //! it doesnt matter what is written to this register
        outportb( (dma==0) ? DMA0_CLEARBYTE_FLIPFLOP_REG : DMA1_CLEARBYTE_FLIPFLOP_REG, 0xff);
}
```

Reset

In a very similar fashion, you can reset the a DMAC by writing any value to the following registers:

ISA DMAC Reset Ports	
Port	Descripton
0x0D	DMAC 0 (16 bit) Slave (write)
0xD8	DMAC 1 (8-bit) Master (write)

For example, assuming **DMA0_TEMP_REG** is 0x0D:

```
void dma_reset (int dma){

        //! it doesnt matter what is written to this register
        outportb(DMA0_TEMP_REG, 0xff);
}
```

Unmask All Registers

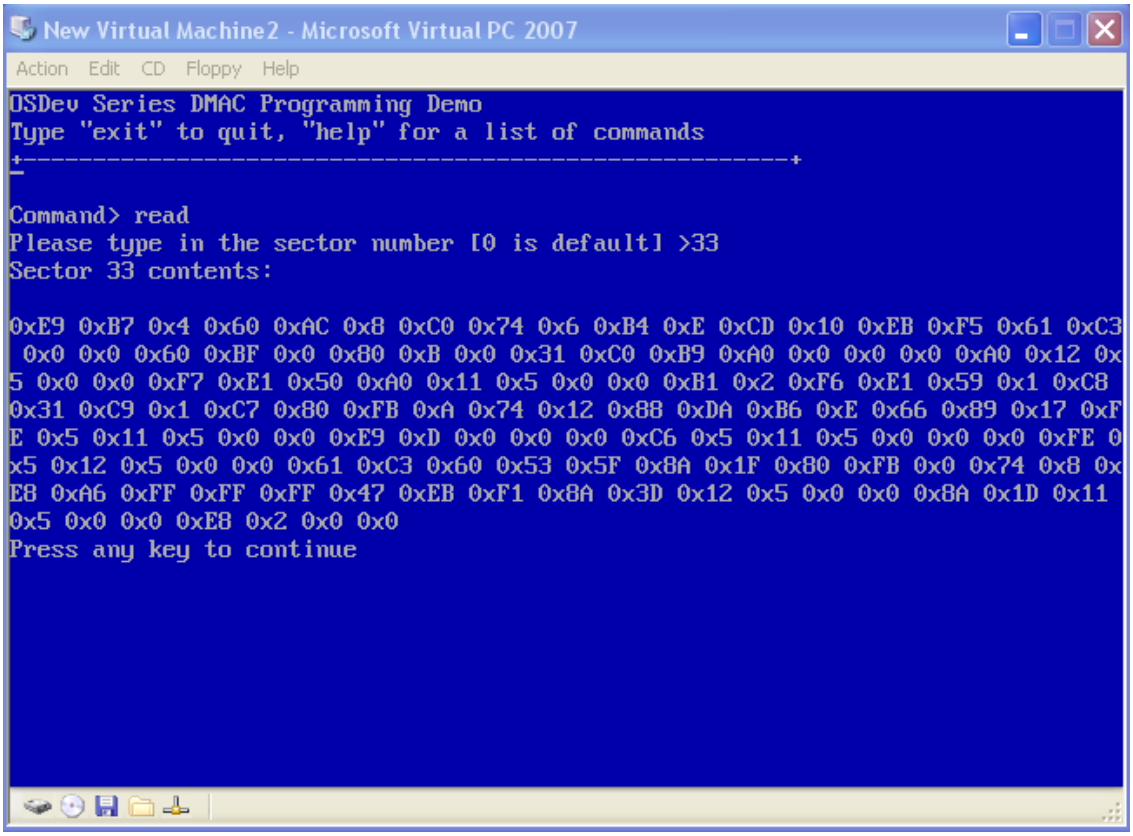
In yet another similar fashion, the same concept applies with this command! Wouldn't it be great if all hardware programming commands were this easy? ;)

ISA DMAC UnMask All Ports	
Port	Descripton
0x0E	DMAC 0 (16 bit) Slave (write)
0xDC	DMAC 1 (8-bit) Master (write)

So, assuming **DMA1_UNMASK_ALL_REG** is 0x0E, this will unmask all registers from the slave DMAC:

```
void dma_unmask_all (int dma){

        //! it doesnt matter what is written to this register
        outportb(DMA1_UNMASK_ALL_REG, 0xff);
}
```

Demo



Demo running in Virtual PC
Demo Download

Yey, its time for another demo!! The bad news is that this demo looks exactally the same as the last chapter (Except it now works with both Bochs and Virtual PC). The good news is that it has been upgraded to use our new DMA interface.

The core of the new code is found in the HAL - **dma.h** and **dma.cpp** which containes all of the code from this chapter. There is one minor change, however. Because the AUTOINIT bit in the Mode register for the DMAC is not well supported in Virtual PC, our **dma_set_read** and **dma_set_write** routines do NOT set the bits in the demo code:

```
//! prepares channel for read
void dma_set_read (uint8_t channel) {

	dma_set_mode (channel,  DMA_MODE_READ_TRANSFER | DMA_MODE_TRANSFER_SINGLE);
}

//! prepares channel for write
void dma_set_write (uint8_t channel) {

	dma_set_mode (channel,
		DMA_MODE_WRITE_TRANSFER | DMA_MODE_TRANSFER_SINGLE);
}
```

During a read sector operation, the floppy driver's **flpydsk_read_sector_imp** routine initializes the DMAC, and prepares the DMAC for a read operation. The rest of the routine (which has been edited out) is the same from the last chapter and is responsible for sending the READ command to the FDC. **DMA_BUFFER** is just a buffer of free memory that can be used for DMAC transfers. It is the same from the last chapter. **dma_initialize_floppy** initializes the DMAC using our new DMA minidriver to prepare it for use by the floppy driver. (We will look at that shortley.) After initializing the DMAC, we prepare the DMAC for our READ operation by calling our drivers **dma_set_read** routine on channel **FDC_DMA_CHANNEL**. **FDC_DMA_CHANNEL** is channel 2 (Remember that the FDC uses channel 2 on the DMAC?)

```
//! read a sector
void flpydsk_read_sector_imp (uint8_t head, uint8_t track, uint8_t sector) {

	uint32_t st0, cyl;

	//! initialize DMA
	dma_initialize_floppy ((uint8_t*) DMA_BUFFER, 512 );

	//! set the DMA for read transfer
	dma_set_read ( FDC_DMA_CHANNEL );

	//! rest of the code is the same...
}
```

dma_initialize_floppy is responsible for preparing the DMAC using our new minidriver for use by the floppy driver. This is where all of the fun stuff is at!

We first reset the master DMAC by calling **dma_reset()**. We then disable (mask) channel 2 (used by the FDC) by calling **dma_mask_channel()**. This insures the channel is no longer in use so that we can modify it.

Now for the fun stuff. To set the address the channel will use, we call our **dma_set_address** routine. This allows us to set the low and high parts of the address to the channel. We use an union to make it a little easier to access the byte components of the members. That is, we set a.l to the buffer that the channel will use. Thanks to the union, a.byte[0] now refers to the low byte of that value, byte[1] referrs to the second byte, etc. We do the same for the **length** which is the size of the buffer. So we set the buffers address by calling **dma_set_address** with the low and high bytes of the uhm.. buffer address and the length by calling **dma_set_count** in the same way. Cool, huh?

Okay, okay, thats all fine and all but whats with the **dma_reset_flipflop** calls? The flipflop is only used with the 8 bit DMAC when working with 16 bit data. If we were working with the 16 bit DMAC, we would not need to call it. The flipflop is used to select between the high and low bytes of the 16 bit data. **When you reset the flipflop, you are telling the DMAC that the next byte data will be the low byte.** If the flipflop is not in its default position, it will be selected as the high byte. This has to be selected because the DMAC is working with 16 bit data on an 8 bit data bus. How does it know what part of the 16 bit data this byte refers to?

Finally we set the DMAC for a read operation by calling **dma_set_read()** and unmasking all of the channels so that they can be used by devices again. This is important as it allows the FDC to use channel 2 on the DMAC.

```
bool _cdecl dma_initialize_floppy(uint8_t* buffer, unsigned length){
   union{
      uint8_t byte[4];//Lo[0], Mid[1], Hi[2]
      unsigned long l;
   }a, c;

   a.l=(unsigned)buffer;
   c.l=(unsigned)length-1;

   //Check for buffer issues
   if ((a.l >> 24) || (c.l >> 16) || (((a.l & 0xffff)+c.l) >> 16)){
#ifdef _DEBUG
      _asm{
         mov      eax, 0x1337
         cli
         hlt
      }
#endif
      return false;
   }

   dma_reset (1);
   dma_mask_channel( FDC_DMA_CHANNEL );//Mask channel 2
   dma_reset_flipflop ( 1 );//Flipflop reset on DMA 1

   dma_set_address( FDC_DMA_CHANNEL, a.byte[0],a.byte[1]);//Buffer address
   dma_reset_flipflop( 1 );//Flipflop reset on DMA 1

   dma_set_count( FDC_DMA_CHANNEL, c.byte[0],c.byte[1]);//Set count
   dma_set_read ( FDC_DMA_CHANNEL );

   dma_unmask_all( 1 );//Unmask channel 2

   return true;
}
```

Conclusion

Well, another chapter down, huh? This chapter wasnt as complex or hard as some of the ealier chapters, so its a nice break isnt it?

From here we cannot get much further without the capability of loading files from disk. We have the ability of loading data from disk - but not files. This is done through a **File System Driver**. But wait! We have already covered FAT12 like .. 2 times already! Ugh, just goes to show how often we need to rewrite things. Rather then recovering the same material for a third time, I will be adding another subject to the mix: **Virtual File Systems (VFS)**. For a little fun, I may even add the ability of executing a demo program come next chapter :)

With the amount of readers wanting to add a graphical touch to their operating systems, I may also release a few advanced chapters related to Vesa Bios Extensions (VBE) and Video Graphics Array (VGA) / Super VGA (SVGA) as well. As well as turning our current system into a real microkernel - DLL support, drivers, and native PE resources support.

Alot of cool stuff coming up :) If you have any ideas of topics that you would like us to cover, feel free to let me know.

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

References

- 82C37A CMOS High Performance Programmable DMA Controller datasheet
- *"The Undocumented PC"*

Questions or comments? Feel free to Contact me.

Chapter 20 Home Chapter 22

Operating Systems Development Series

Operating Systems Development - FileSystems and the VFS

by Mike, 2010

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

Welcome to the 22'th chapter in a never-ending series for operating system development! This is more then chapter 22 but also year 2 for the OS Development Series.

This is yet another filesystem related tutorial (Dont worry, its the last one ;)). The first one was needed so we can load our main bootloader program from the bootcode, The second one was for our main boot program so it can load our kernel. Now we need one more for our kernel so our kernel can load programs and execute them. There is a difference between this chapter and the other two, however - this one will be in C instead of assembly language. :)

To spice things up, however, and introduce something new, we will also be looking into **Virtual FileSystems (VFS)**. This will allow us to interface with any filesystem driver and different disk devices in the same way. It can be used for both local disk drives, but can also be used to interface with any network filesystem.

Ready?

File Systems

Abstract

File System

A **File System** defines a logical way to read and write information. In this way, it can be coinsidered a **specification**. Most PC file systems are based off of the desktop concept of files and folders.

There are alot of different kinds of file systems. Some are widley use (Like FAT12, FAT16, FAT32, NTFS, ext (Linux), HFS (Used in older MACs); other filesystems are only used by specific companies for in house use (Like the GFS - Google File System). Some filesystems are used in networking only (NFS). You can also develop and design your own file system implimentation.

File Systems are used for data storage and organizing data. They help provide a straightforward way to access files and directories on removal media (floppies, flash drives, CDs, DVDs), local drives (hard disk drives), and network clients. File Systems can also exist as an in-memory image. For example, you can load a file that containes a "foot print" of a special type of file system in it.

Files and Folders

A **file** is a group of data that represents something to a program or to the user. This data can be anything we want it to be. It all depends on how we interprate the data. For example, a **text file** containes text information. A file can also be an image of something. A **folder** is a logical group of files. It is also known as a **directory**.

Directories provide us a way to manage a large amount of files. Directories typically form a **tree** structure. This is known as a **directory tree**. There is only one directory that is the parent of all directories and files: the **Root Directory**. A **File Path** is the location of a file in the directory tree. For example, the file **a:\myfile.txt**, myfile.txt is the filename. It is in the root directory at the device known as "a:". **a:\mydir\myfile.txt** is a file, myfile.txt, located in the subdirectory, mydir, that is, in turn, located in the root directory on device "a:".

File and Folder Naming

The name of a folder or file is a string representing that file or folder, usually by its contents. File Systems impliment file naming and folder naming differently, and each has their own constraints. For example, FAT12 stores filenames and folder names in a directory entry as an array of 11 bytes (8 for filename, 3 for extension. This is also known as the **8.3 naming convention**) This limits file names and folder names to 11 characters. On the other hand, NTFS is limited to 255 characters with **Long File Name (LFN)** support. NTFS, for another example, stores file names along with file attributes in a **Master File Table**.

Most filesystems file names are not case sensitive. However, some filesystems may store filenames differently internally. For example, you may have found out that you can have an 8.3 lowercase file name for a file on the floppy disk, but be able to load the file from your OS by using an all uppercase file name. Windows displays the LFN of the file name, while FAT12's 8.3 file entry only containes its 8.3 all-uppercase file name. This is what makes it possible.

File Types

Symbolic link's

Symbolic links are a way to provide shorten paths. For example: a:/folder/link.lnk points to a:/otherfolder/subfolder/subsubfolder/yet another folder/link.txt. Now you can access the text file easily. Symbolic links are also

very often used to make folder organized. Like the Windows Start Menu. Contains symblic links to your programs. A symbolic link is not very hard to implement. You find the node given (which is the link). It seems to be a link, so you get the real path and read that file instead.

Windows Shortcuts are a type of symbolic link.

Pipes

A type of **InterProcess Communication (IPC)** is called a pipe. A pipe is a **virtual file**, usually between two or more processes. The best example may be stdout, stdin and stderror on Unix. They are handled as normal files, but the data written to stdout show up onto the screen (or in stdout.txt).

Special File Types

Metafiles

Some filesystems also impliment special files and folders specifically for filesystem use. Typically you cannot have two files or folders with the same name (nor a filename sharing the same name as a folder) in the same directory. Because of this, naming a file or folder with one of these hidden files may also not be possible depending on implimentation.

For example, NTFS provides several metafiles for filesystem use. These files are located on the root directory of the system drive (typically C:). $MFT,$MFTMirr, and $LogFile are a few of these files. While they do not ever show up even when view hidden and system files are checked, watch what happens when you create a file with one of the above names there. You can create those files anywhere else, but you will get a "file already exists" error when creating one on the root directory do to the metafiles.

Device Files

Unix-like systems, DOS (and, in turn, Windows) has **Device Files** which are special "files" that represent a device. For example, NUL (null device), CLOCK$, PRN (printer), etc. Here is the list of device files:

- CON
- PRN
- AUX
- CLOCK$
- NUL
- COM0, COM1, ... COM9
- LPT0, LPT1, ... LPT9

Because these names have special meaning in DOS and Windows, you cannot name a file or folder any of the above names.

. and ..

. and .. are special files some file systems impliment. '.' is the file name of a file that containes file information that referrs to the current directory. '..' is the file name of a file that containes information that referrs to the parent directory of that file. For example, if there is a file located at **c:\mydir\file.txt**, and **c:\mydir** was the current directory, the pathname **..** will refer to C: while the pathname **.** will refer to c:\mydir.

File System Types

Flat File Systems

A **Flat File System** is a filesystem that does not support subdirectories. Instead, all of the files are in the same (root) directory. Many early computer systems used flat file systems. Modern operating systems typically impliment more advanced hierarchical file systems. While small and easy to impliment, flat file systems are hard to orginize.

Hierarchical File Systems

This type of file systems supports subdirectories. Most modern file systems (including FAT12,FAT16, FAT32,etx,NTFS) fit into this categary. (The first version of FAT12 was a flat file system. Later versions support subdirectories however.)

Journaling File Systems

This type of file system uses a "journal" of file system changes. This is a log of changes the system intends to make to files or directories prior to completeing the steps. This insures that, if a crash occures during a filesystem operation (like writing a file), the journal can be read to undo the changes made to repair the filesystem.

File System Drivers

While a **file system** defines a specification for reading and writing "files" and "directories", a **file system driver** containes the implimentation of a specific type of file system. A good example of a file system driver is **ntfs.sys** which containes Microsoft's implimentation of the NTFS File System. File system drivers are also sometimes implemented as minidrivers inside of larger software. Bootloaders are a good example. Because boot loaders have to be able to load files from disk without a separate driver program, they contain several filesystem minidrivers for different types of filesystems inside of the bootloader itself. If you developed the bootloader in the series, you have already experienced the FAT12 file system and developed a FAT12 minidriver for our bootloader.

Virtual FileSystem (VFS)

Abstract

A **Virtual File System (VFS)** is an abstraction layer ontop of specific filesystem implimentations. The software accesses storage devices through a VFS. This allows the software to read or write to different storage devices without any knowledge of the device or filesystem that is being used. It also allows the same code to work with any number of installed filesystems or devices.

The basic idea is to allow a single system interface to work with any filesystem in a uniform way. Windows, Linux, and Mac OS all support VFS in different ways.

Implimentation

There are different ways to impliment a VFS.

Mount Point List

A **mount point list** is a list of mounted file systems and where they are mounted. For example, if a file needs to be read from, the OS typically calls the VFS ReadFile() function which searches through the list of mounted file systems to locate the device and file system the file is in. It then passes the read request to that file system's ReadFile() function.

Node Graph

A **Node Graph** contains a graph of nodes that represent files of different types: files, folders, mount points, etc. Each file node structure typically containes function pointers to file system-specific routines for reading and writing files.

For example, we can create a FILE structure like this:

```
typedef struct _FILE {

        char        name[32];    //filename
        uint32_t    flags;       //flags
        uint32_t    fileLength;  //length of file
        read_funct  read;        //function pointers to read,write,open,close file
        write_funct write;
        open_funct  open;
        close_funct close;

}FILE, *PFILE;
```

Notice the function pointers are stored in this FILE structure. Lets say we want to read a file, so we call fopen(), which, eventually, calls our VFS OpenFile() function. All the VFS file operation routines ever need to do is pass control to that specific FILE's function pointers:

```
void VfsOpenFile (PFILE file, const char* filename) {

        if (file)
                file->open (filename);

}
```

This allows the filesystem-defined routine to be called.

DOS and Windows

DOS and Windows assigns a letter from 'a' through 'z' to represent a mounted file system. Windows keeps a symbolic link between a drive letter and its Object Manager name. For example, the drive letter c: (symbolc link name \\GLOBAL??\C:) may be mapped to the Object name \Device\HardDiskVolume1 device object. A File System can register themself to own a device object. If a file system is found to own the object, the rest of the file path name ("myfile.txt" in this example) is passed to that filesystem's FileOpen() function.

Drive letter assignment

Windows supports assigning drive letters to devices and partitions representing mounted file systems. (During boot, if no filesystem driver registers to own a device object, Windows uses its RAW minidriver for the devices.) Drive letters can also refer to network shared drives, virtual disk images, or a symbolic link to another location in the local or a network client. However, they are limited to 26 devices do to only 26 letters that can be used from 'a' to 'z'.

Interface

For simplicity, we will be using drive letter assignment along with a mount point list in our VFS implimentation. Our implimentation needs to be simple because we do not have device management nor I/O management in the OS presented in the series.

I personally recommend developing the VFS first prior to the filesystem driver. This way the interface and framework of the VFS will have already been completed.

FILE

Anyone that has used C is already famalier with the infamous FILE* data type. FILE* is an **Abstract Data Type (ADT)** that represents a pointer to a file object. ISO C defines that C implimentations must define a FILE type, however does not define what

is inside of the structure. That is, while FILE* is ISO C, the structure contents is implimentation-defined.

We can define a file structure that will represent the current state of a file any way we want. So lessee... a file has a name and a size, so thats two members already. We need a way to flag if its the **End of File (EOF)**, and file-specific flags, so thats two more members. We also need a way to keep track of a files current position (its cluster and the clusters offset), and now we have something like this:

```
typedef struct _FILE {

        char        name[32];
        uint32_t    flags;
        uint32_t    fileLength;
        uint32_t    id;
        uint32_t    eof;
        uint32_t    position;
        uint32_t    currentCluster;
        uint32_t    device;

}FILE, *PFILE;
```

That was easy, huh? **id** can be used for idenitification purposes if you like. **device** represents the device the file resides on.

Types of files

There are alot of different types of files that we have talked about: files, directories, symbolic links, etc. For simplicity, we will only focus on files and directories. These will be used in the **flags** member of our FILE structure above to represent the type of file.

```
#define FS_FILE       0
#define FS_DIRECTORY  1
#define FS_INVALID    2
```

Operations

There are some typical operations we can perform on a file:

- Open
- Close
- Read
- Write
- Mount
- Unmount

Open and Close operations perform opening and closing a file object (file or directory, whatever the file type is), while reading and writing operations perform reading and writing the file type. All of these are exposed to the programmer through the standard C file I/O functions.

For our VFS, they are exposed through a **Volume Manager** located in fsys.h:

```
extern FILE volOpenFile (const char* fname);
extern void volReadFile (PFILE file, unsigned char* Buffer, unsigned int Length);
extern void volCloseFile (PFILE file);
extern void volRegisterFileSystem (PFILESYSTEM, unsigned int deviceID);
extern void volUnregisterFileSystem (PFILESYSTEM);
extern void volUnregisterFileSystemByID (unsigned int deviceID);
```

For example, lets say we call the C fopen() routine. That will call our volOpenFile() routine which returns a FILE object. We passed a path to the file, like "a:\myfile.txt". The Volume Manager indixes into the mount point list and verifies that a file system has registered for the device ID that represents 'a'. If it has, it calls that filesystem drivers FileOpen() method passing "myfile.txt". Dont worry if it sounds complicated. It can be; but the design of how its implemented in the demo is very easy.

Volume Manager Implimentation

File System Abstraction

The first thing we need is a way to abstract filesystem-specific information. This includes the name of the filesystem and the operations that can be performed on files. This is done using function pointers.

```
typedef struct _FILE_SYSTEM {

        char Name [8];
        FILE                (*Directory)   (const char* DirectoryName);
        void                (*Mount)       ();
        void                (*Read)        (PFILE file, unsigned char* Buffer, unsigned int Length);
        void                (*Close)       (PFILE);
        FILE                (*Open)        (const char* FileName);

}FILESYSTEM, *PFILESYSTEM;
```

Implimentation

The Volume Manager impliments our VFS in the demo. Its in the files fsys.h and fsys.cpp. Remember that we will be using drive letter assignment to represent devices? Because there are 26 possible devices, it is helpful to make a constant, **DEVICE_MAX**. Because each device can only have one mountable file system, we store them in a list (like a mount point list).

```
#define DEVICE_MAX 26

//! File system list
PFILESYSTEM _FileSystems[DEVICE_MAX];
```

Here is how it works. Because we are storing the filesystems as a list of pointers, if a pointer is valid, the filesystem has been registered there. Each element in the array represent the drive letter that it refers to. So 'a' is ar _FileSystems[0], 'b' is at _FileSystems[1], etc. It is the filesystems responsibility to manage the disk that they are writing on.

Using this method provides a very basic but easy way of accessing devices. For example volOpenFile() only needs to check the first character of the path (the drive letter) and do a lookup into the list to see if a filesystem is registered for that device. If it is, it can call that filesystem's open() method and pass the filename to the driver. We default to using 'a', however if the input path contains an ':' then we use the first character for the device instead. This allows us to call volOpenFile in two ways: passing a string like **"myfile.txt"** and **"a:myfile.txt"**, where "a" is the device the file is in. Cool, huh?

```
FILE volOpenFile (const char* fname) {

	if (fname) {

		//! default to device 'a'
		unsigned char device = 'a';

		//! filename
		char* filename = (char*) fname;

		//! in all cases, if fname[1]==':' then the first character must be device letter
		if (fname[1]==':') {

			device = fname[0];
			filename += 2; //strip it from pathname
		}

		//! call filesystem
		if (_FileSystems [device - 'a']) {

			//! set volume specific information and return file
			FILE file = _FileSystems[device - 'a']->Open (filename);
			file.deviceID = device;
			return file;
		}
	}

	FILE file;
	file.flags = FS_INVALID;
	return file;
}
```

All of the other file operation routines are basically the same. Knowing how our VFS is storing filesystems, you can probably guess how volRegisterFileSystem() family of routines work. All they basically do is store a pointer to the filesystem in the list or clear it.

```
void volRegisterFileSystem (PFILESYSTEM fsys, unsigned int deviceID) {

	if (deviceID < DEVICE_MAX)
		if (fsys)
			_FileSystems[ deviceID ] = fsys;
}
```

Alright then! So we initialize the filesystem driver, which calls VolRegisterFileSystem() to register itself. We call fopen(), which calls VolOpenFile(), which in turn calls our filesystem's open() method. Everything is now in place but we are missing something... something very important... the filesystem driver itself!

Right, I suppose we should go into it .. again...

FAT12 - Take Three

Introduction

We have looked at and implemented FAT12 two times in the past throughout the series. Because of this, I do not plan on covering FAT12 in great detail again. However, this will be a review of FAT12 along with the C driver code and how it works.

If needed, please reference Chapter 11 while reading.

Boot Sector

Remember that alot of important filesystem information is stored in the boot sector along with our boot strap program? More specifically, it is located in the **Bios Paramater Block (PBP)** located in the boot sector.

When we mount our filesystem, we will need to read from the BPB and store this information for later use. To do this, we can create a structure that matches the boot sector:

```
typedef struct _BOOT_SECTOR {

        uint8_t                 Ignore[3];              //first 3 bytes are ignored (our jmp instruction)
        BIOSPARAMATERBLOCK      Bpb;                    //BPB structure
        BIOSPARAMATERBLOCKEXT   BpbExt;                 //extended BPB info
        uint8_t                 Filler[448];            //needed to make struct 512 bytes

}BOOTSECTOR, *PBOOTSECTOR;
```

A good example of what the boot sector looks like is to think about what our Stage 1 Bootloader program looks like in memory. The very first instruction in Stage1 (Please see Chapter 4's demo, Stage1.asm) was **jmp loader**. This is a three byte instruction, so the first 3 bytes in the above structure is the **Operation Code (OPCode)** of our jmp instruction.

Also remember from Chapter 4 that we have covered the OEM Paramater Block (aka, Bios Paramater Block (BPB). The BPB is located right after our 3 byte jump instruction. Because of this, the BIOSPARAMATERBLOCK is next in this structure. I also provide the BIOSPARAMATERBLOCKEXT structure which is an extension to the BPB for some other file systems, such as FAT32.

The last 448 bytes of the bootsector contain the rest of our boot sectors program code. Because its not important to us right now, we just treat it as padding in the **Filler** member. This insures the BOOTSECTOR structure is exactally the same size as our on-disk boot sector (512 bytes).

BIOSPARAMATERBLOCK is a structure that defines the format for a BPB. It is the same structure that is in the boot sector and has been covered in more depth in Chapter 5.

```
typedef struct _BIOS_PARAMATER_BLOCK {

        uint8_t                 OEMName[8];
        uint16_t                BytesPerSector;
        uint8_t                 SectorsPerCluster;
        uint16_t                ReservedSectors;
        uint8_t                 NumberOfFats;
        uint16_t                NumDirEntries;
        uint16_t                NumSectors;
        uint8_t                 Media;
        uint16_t                SectorsPerFat;
        uint16_t                SectorsPerTrack;
        uint16_t                HeadsPerCyl;
        uint32_t                HiddenSectors;
        uint32_t                LongSectors;

}BIOSPARAMATERBLOCK, *PBIOSPARAMATERBLOCK;
```

The above structure should look familier :) If not, please read its description in Chapter 5

BIOSPARAMATERBLOCKEXT, however, may be new. While we have already covered the BPB in depth and used it in the past for FAT12 parsing, FAT12 bootsectors do not rely on the BPB extended members. FAT32, however, does.

```
typedef struct _BIOS_PARAMATER_BLOCK_EXT {

        uint32_t                        SectorsPerFat32;    //sectors per FAT
        uint16_t                        Flags;              //flags
        uint16_t                        Version;            //version
        uint32_t                        RootCluster;        //starting root directory
        uint16_t                        InfoCluster;
        uint16_t                        BackupBoot;         //location of bootsector copy
        uint16_t                        Reserved[6];

}BIOSPARAMATERBLOCKEXT, *PBIOSPARAMATERBLOCKEXT;
```

Thats everything :) There is nothing special here-everything has already been covered in detail in previous chapters. These structures provide the filesystem driver an easy way of referencing data in the BPB for later filesystem use. All we need to do is read in the bootsector, and accessing the data through a PBOOTSECTOR. :)

We read the sector using our floppy disk driver that we developed in the previous chapter.

```
        //! Boot sector info
        PBOOTSECTOR bootsector;

        //! read boot sector
        bootsector = (PBOOTSECTOR) flpydsk_read_sector (0);
```

That is all that is needed :) All of our important information is now in **bootsector.bpb**. All thats needed is mounting the filesystem...

Mounting the filesystem

Now that we have our BPB information in memory, we need to prepare the filesystem for use. We start this by first deciding what information we need.

Okay.. let see, we will need to total number of sectors on disk. We will also need to know the total number of directory entries. Other helpful information can be for use with the **File Allocation Table (FAT)** and the Root Directory:

```
typedef struct _MOUNT_INFO {

        uint32_t numSectors;
        uint32_t fatOffset;
        uint32_t numRootEntries;
        uint32_t rootOffset;
        uint32_t rootSize;
        uint32_t fatSize;
        uint32_t fatEntrySize;

}MOUNT_INFO, *PMOUNT_INFO;
```

Okay... Remember that we already have the bootsector stored in our BOOTSECTOR structure? Knowing this, we can simply copy over some of the information from the BPB to our MOUNT_INFO structure.

Alright.. Lets locate the location of the first FAT and root directory in a FAT12 formatted disk:

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

Notice that there are two FATs. The first FAT is right after the boot sector on disk. Because of this, we set **fatOffset** in MOUNT_INFO to 1. Also note that the Root Directory is right after both FATs. Knowing this, we can come up with a simple calculation to find the starting sector of the root directory. **(NumberOfFATs * sectorsPerFAT) + 1**. We need to add 1 for the bootsector.

We now have the location of the first FAT and root directory. To find the size of the root directory, all we need is the number of entries in the root directory and the size of each entry. Each directory entry in FAT12 is a specific structure format that is 32 bytes in size. So all we have to do is **bootsector->Bpb.NumDirEntries * 32**. This is the number of bytes the directory takes up. We divide it by the bytes per sector to convert it to a sector count.

```
        //! store mount info
        _MountInfo.numSectors     = bootsector->Bpb.NumSectors;
        _MountInfo.fatOffset      = 1;
        _MountInfo.fatSize        = bootsector->Bpb.SectorsPerFat;
        _MountInfo.fatEntrySize   = 8;
        _MountInfo.numRootEntries = bootsector->Bpb.NumDirEntries;
        _MountInfo.rootOffset     = (bootsector->Bpb.NumberOfFats * bootsector->Bpb.SectorsPerFat) + 1;
        _MountInfo.rootSize       = ( bootsector->Bpb.NumDirEntries * 32 ) / bootsector->Bpb.BytesPerSector;
```

That is all that there is to it. We have our FAT12 driver initialized. Easy, huh? We have the important filesystem information in MOUNT_INFO so all thats needed is to parse the directories and load a file. :)

Directory parsing

Format

A directory in FAT12 is composed of 32 byte structures that provide information about a file or subdirectory. Each directory entry has the following format:

```
typedef struct _DIRECTORY {

        uint8_t   Filename[8];          //filename
        uint8_t   Ext[3];               //extension (8.3 filename format)
        uint8_t   Attrib;               //file attributes
        uint8_t   Reserved;
        uint8_t   TimeCreatedMs;        //creation time
        uint16_t  TimeCreated;
        uint16_t  DateCreated;          //creation date
        uint16_t  DateLastAccessed;
        uint16_t  FirstClusterHiBytes;
        uint16_t  LastModTime;          //last modification date/time
        uint16_t  LastModDate;
        uint16_t  FirstCluster;         //first cluster of file data
        uint32_t  FileSize;             //size in bytes

}DIRECTORY, *PDIRECTORY;
```

That is all that there is to it :) This is a directory entry - the information stored in our DIRECTORY structure can be a subdirectory or a file. **Filename** and **Ext** containes the file or directories 8.3 format name.

Attrib containes the attributes of a file or directory. It has the following values for reference:

- Read only: 1
- Hidden: 2
- System: 4

- Volume Lable: 8
- Subdirectory: 0x10
- Archive: 0x20
- Device: 0x60

Please note that we will not be using this in the series as it is not needed. However, you can provide support for working and setting file attributes in your own system if you like.

All **date** members in this structure follow a specific bit format:

- **Bits 0-4**: Day (0-31)
- **Bits 5-8**: Month (0-12)
- **Bits 9-15**: Year

All **time** members in this structure follow a specific bit format:

- **Bits 0-4**: Second
- **Bits 5-10**: Minute
- **Bits 11-15**: Hour

Because we have no need in modifying or retrieving file or directory date or time information we are not using them in the series. However, I encourage our readers to add the functionality themselves later on it they like.

Remember that, for a FAT12 formatted floppy disk, a cluster is the same size as a sector (512 bytes). Because of this, the **FirstCluster** field in DIRECTORY also points to the first sector of a file. Thus, by reading this sector, you effectively read the first 512 bytes of the file.

Now lets parse our directory and find our file...

Parsing

Remember that a directory containes a list of directory entry structures. Knowing this, parsing the directory to find a file or directory becomes very easy.

We begin with loading the root directory. Remember that we retrived the root directory sector from the BPB when we mounted the filesystem and stored it into **_MountInfo.rootOffset**. Thus, all we need to do is to load the sector, and use a **DIRECTORY*** to access the directory entries.

Then we loop and compare filenames to find a match. We convert the input filename to its DOS 8.3 filename format using ToDosFileName(). For example, turning the input filename of "Myfile.txt" to the FAT12 internal format "MYFILE TXT".

We read in a sector and compare each entry in the sector. You will also notice that we turn the filenames into C strings so we can use a simple strcmp() call to test if filenames match. When we found a match, we fill out our FILE structure and return it.

Lets take a look:

```
FILE fsysFatDirectory (const char* DirectoryName) {

	FILE file;
	unsigned char* buf;
	PDIRECTORY directory;

	//! get 8.3 directory name
	char DosFileName[11];
	ToDosFileName (DirectoryName, DosFileName, 11);
	DosFileName[11]=0;
```

DirectoryName contains the directory or file name we are wanting to find. We convert the input filename, like "myfile.txt" into its DOS 8.3 filesystem format "MYFILE TXT" and store it in DosFileName.

```
	for (int sector=0; sector<14; sector++) {

		//! read in sector
		buf = (unsigned char*) flpydsk_read_sector ( _MountInfo.rootOffset + sector );

		//! get directory info
		directory = (PDIRECTORY) buf;
```

We are reading from the root directory. The root cluster is stored in _MountInfo, which containes information obtained from the **Bios Paramater Block (BPB)** when the file system was mounted. **_MountInfo.rootOffset** contains the first cluster of the root directory. The root directory contains, at most, 224 DIRECTORY entries. A DIRECTORY entry is 32 bytes, 224*32=7168 bytes, 7168 bytes / 512 bytes (512 bytes in a cluster) = 14. This means the root directory consists of 14 clusters.

Knowing this, rather then loading the entire directory at once, we can load it sector by sector and parse each part of the directory.

```
		//! 16 entries per sector
		for (int i=0; i<16; i++) {

			//! get current filename
			char name[11];
```

```
                        memcpy (name, directory->Filename, 11);
                        name[11]=0;

                        //! find a match?
                        if (strcmp (DosFileName, name) == 0) {
```

Knowing that a DIRECTORY entry is 32 bytes, 512 bytes per cluster / 32 bytes = 16. This means there are 16 DIRECTORY entries in one sector. So, we loop through each entry and compare filenames to locate the file or directory that we are looking for. If they match, we can create a new **FILE** object and return it. **file.currentCluster** will contain the first cluster of the file for reading later, **file.fileLength** contains the size of the file, in bytes. **directory->Attrib** contains the files attributes. We set it based on its DIRECTORY entry attribute.

```
                                //! found it, set up file info
                                strcpy (file.name, DirectoryName);
                                file.id             = 0;
                                file.currentCluster = directory->FirstCluster;
                                file.eof            = 0;
                                file.fileLength     = directory->FileSize;

                                //! set file type
                                if (directory->Attrib == 0x10)
                                        file.flags = FS_DIRECTORY;
                                else
                                        file.flags = FS_FILE;


                                //! return file
                                return file;
                        }
```

Almost there... If we have not found the file or directory yet, we just move onto the next DIRECTORY entry. If we never find the file, we set FS_INVALID and return.

```
                        //! go to next directory
                        directory++;
                }
        }

        //! unable to find file
        file.flags = FS_INVALID;
        return file;
}
```

Thats it! The above routine works for directories and files in FAT12. By calling it, it will search the root directory for any folder or file name and return its information.

SubDirectories

While the old version of FAT12 was flat, new versions of this file system supports subdirectories. This allows us to be able to use directories and manage a lot of files more easily. For example, it would be a good idea in a large OS to separate OS-specific files in a **system** directory, or a **user** directory containing user profiles.

A subdirectory is just an ordinary file with the DIRECTORY flag set. Because of this, we first need to know how to read files so lets look at that now.

File Reading

Format

Okay, so we can now parse directories and locate files. We now need a way to read the file's contents. Remember that, technically, we can already read the first 512 bytes of any file by just the **FirstCluster** field in the directory entry structure for that file. To read more then one cluster, we have to parse the **File Allocation Table (FAT)**.

Recall that FAT consists of a number of entries containing cluster numbers. The size of these entries depends on the filesystem. FAT12 has 12 bits per entry, FAT16 has 16 bits, FAT32 has 32 bits per entry.

Think of the FAT as - not as a linked list, but rather a table of entries that represent the whole physical disk. The first cluster of the disk is represented by the first entry of the FAT. The second cluster is represented by the second entry, and so on. This means there is a one to one relationship between a cluster and a FAT entry. This makes reading and writing files in FAT12 easy.

Reading a file

To read a file, we just read the current cluster of the file. We try to locate its next cluster on disk by parsing the FAT table. After we find the next cluster, update the "current cluster" for the next file read.

The cluster to read was set when the file is opened. On the first call to this routine, **file->currentCluster** is the same as **DIRECTORY->FirstCluster**.

This cluster is an offset into the data area on disk. Lets recall the format of a FAT12 formatted disk and locate our FAT and data area:

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

Remember that each FAT takes 9 sectors. Because there are two FATs, 9+9=18. We have also concluded that our root directory is 14 sectors in the previous section. 18+14=32. This is the amount of sectors both FATs and the root directory take up. So far our equation is **32 + file->currentCluster**. We need to subtract 1 and we have **32 + (file->currentCluster - 1)**. This is the sector to read in and containes the file data.

```
void fsysFatRead(PFILE file, unsigned char* Buffer, unsigned int Length) {

        if (file) {

                //! starting physical sector
                unsigned int physSector = 32 + (file->currentCluster - 1);

                //! read in sector
                unsigned char* sector = (unsigned char*) flpydsk_read_sector ( physSector );

                //! copy block of memory
                memcpy (Buffer, sector, 512);
```

To read in the next cluster we have to parse the FAT tables. Because a FAT table is 9 sectors, rather then reading all 9 sectors we determin what sector we need to read.

We first get a byte offset into where the next cluster will be in. To do this, we multiply the cluster value by the size of a cluster. This gets stored in **FAT_Offset**. The size of a FAT32 cluster is 4 bytes, so we would mutiply by 4 is we are using FAT32. We would multiply by 2 if we were using FAT16 as that uses 2 bytes per cluster entry. Thats all fine of course, but what about FAT12? FAT12 uses 12 bits per cluster entry. Thats 8 bits (for the 1st byte) and 4 bits (for the 2nd byte. 4 bits is half of 8 bits, so its 0.5) so its 1.5 bits per cluster entry.

After this, just divide this byte offset by the size of a sector to obtain the sector of the FAT to read in. The remander is the offset in this sector, which is the cluster to read from the FAT. This is in **entryOffset**.

FAT is defined as **uint8_t FAT [SECTOR_SIZE*2]**. Notice that we read 2 sectors of our FAT into memory instead of one. Why do this? Knowing a sector size is 512 bytes, 512 bytes * 8 = 4096 bits per sector. 4096 bites / 12 bits (for a FAT entry) and we have 341.3333..etc. This means that an entry will sit between the 1st and 2nd sector. This will cause problems when loading files. Because of this, we have to load an additional sector so the last cluster value of the 1st sector will not be corrupt.

```
                unsigned int FAT_Offset = file->currentCluster + (file->currentCluster / 2); //multiply by 1.5
                unsigned int FAT_Sector = 1 + (FAT_Offset / SECTOR_SIZE);
                unsigned int entryOffset = FAT_Offset % SECTOR_SIZE;

                //! read 1st FAT sector
                sector = (unsigned char*) flpydsk_read_sector ( FAT_Sector );
                memcpy (FAT, sector, 512);

                //! read 2nd FAT sector
                sector = (unsigned char*) flpydsk_read_sector ( FAT_Sector + 1 );
                memcpy (FAT + SECTOR_SIZE, sector, 512);
```

After the FAT sector has been read, we read in the cluster number.

Now we run into a problem. If we read an 8 bit value, we will not be able to read the whole 12 bits of a cluster value. So, we read 16 bits instead using an uint16_t. Of course, now we have the problem of having to much bits of our 12 bit value.

Lets take a look closer. Lets say this is our FAT. We will separate our FAT into bytes but mark out the 12 bit entries. (This is taken from Chapter 6)

```
                Note: Binary numbers seperated in bytes.
                Each 12 bit FAT cluster entry is displayed.

                                    |                             |
 01011101       0111010   01110101  00111101  0011101  0111010  0011110 0011110
 |              |                   |              |             |              |
 |              |1st cluster        |              |3rd cluster- |              |
 |-0 cluster ----|                  |2nd cluster---|             |4th cluster---|
```

Notice all even clusters accopy all of the first byte, but part of the second. Also notice that all odd clusters occopy a part of their first byte, but all of the second!

With this in mind, this means if the cluster is even, **Mask out the top 4 bits, as it belongs to the next cluster.** If the cluster is odd, **shift it down 4 bits (to discard the bits used by the first cluster.)**

Now that we have all of that out of the way, lets finish off this function:

```
                //! read entry for next cluster
                uint16_t nextCluster = *( uint16_t*) &FAT [entryOffset];

                //! test if entry is odd or even
                if( file->currentCluster & 0x0001 )
                        nextCluster >>= 4;      //grab high 12 bits
                else
```

```
                        nextCluster &= 0x0FFF;   //grab low 12 bits

                //! test for end of file
                if ( nextCluster >= 0xff8) {

                        file->eof = 1;
                        return;
                }

                //! test for file corruption
                if ( nextCluster == 0 ) {

                        file->eof = 1;
                        return;
                }

                //! set next cluster
                file->currentCluster = nextCluster;
        }
}
```

Writing a file

[To be completed in the chapter update!]

SubDirectories

A **SubDirectory** is a file with the DIRECTORY attribute set. To read from a subdirectory, all we need to do is locate the FAT12 file on disk with that directory name and read it in the same way as with other files using the FAT.

After the file is loaded, from the first byte to the last is just an array of DIRECTORY entries. Parse the DIRECTORY entries the same way that we did with the root directory to read this directory :-) These will be the files and folders inside of the directory.

Lets take a look:

```
FILE fsysFatOpenSubDir (FILE kFile,
                        const char* filename) {

        FILE file;

        //! get 8.3 directory name
        char DosFileName[11];
        ToDosFileName (filename, DosFileName, 11);
        DosFileName[11]=0;
```

filename contains the file or directory that we want to find. **kFile** is the subdirectory that we want to parse. We convert the input filename, like "myfile.txt" into its DOS 8.3 filesystem format "MYFILE TXT" and store it in **DosFileName**.

```
        //! read directory
        while (! kFile.eof ) {

                //! read directory
                unsigned char buf[512];
                fsysFatRead (&file, buf, 512);

                //! set directort
                PDIRECTORY pkDir = (PDIRECTORY) buf;
```

file is our subdirectory that we want to parse. Remember that it is just an ordinary file in FAT12, so we read in a sector of the file. The file consists of an array of DIRECTORY entries. To make the DIRECTORY members easy to access, we use **pkDir** to point to the sector contents. Now, we search through the directory...

```
                //! 16 entries in buffer
                for (unsigned int i = 0; i < 16; i++) {

                        //! get current filename
                        char name[11];
                        memcpy (name, pkDir->Filename, 11);
                        name[11]=0;

                        //! match?
                        if (strcmp (name, DosFileName) == 0) {
```

Each DIRECTORY entry is 32 bytes. A sector (also cluster in FAT12) is 512 bytes. 512 bytes / 32 bytes = 16 DIRECTORY entries per sector. So, we loop through all 16 entries to compare names. Once we find a filename matching the one we are searching, we have found the file.

```
                                //! found it, set up file info
                                strcpy (file.name, filename);
                                file.id             = 0;
                                file.currentCluster = pkDir->FirstCluster;
                                file.fileLength     = pkDir->FileSize;
                                file.eof            = 0;
```

```
                        //! set file type
                        if (pkDir->Attrib == 0x10)
                                file.flags = FS_DIRECTORY;
                        else
                                file.flags = FS_FILE;

                        //! return file
                        return file;
                }
```

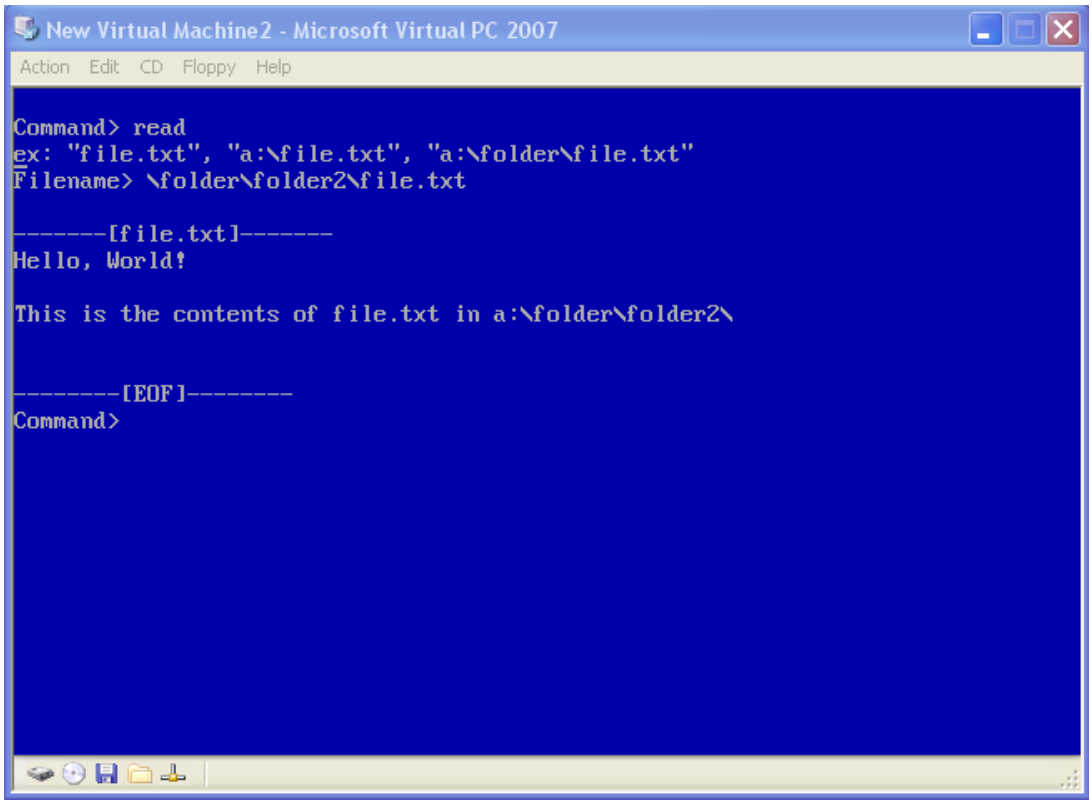
When we have found the file, we fill in our FILE structure - first file cluster (so we can read it later on), file size (so we know when EOF is) and its attribute (file or directory).

If the file has not been found, we just move onto the next entry. This loop will continue until the end of the file. If no file is found, we set FS_INVALID and return.

```
                        //! go to next entry
                        pkDir++;
                }
        }

        //! unable to find file
        file.flags = FS_INVALID;
        return file;
}
```

Notice the simularities between this routine and our **FsysFatDirectory** routine.

Demo



Viewing a file in our OS
DEMO DOWNLOAD

This chapters demo puts everything we covered and impliments a VFS and FAT12 minidriver. It is capable of supporting multiple filesystems, disk devices, subdirectory support, and loading and displaying files.

The demo is also capable of displaying large files and impliments a "press a key to continue" feature for multi-cluster files.

This demo impliments the **strchr()** ISO C routine in our CRT **string.c** to help with text parsing. It also upgrades our **read** command so it is capable of locating and displaying files instead of raw sectors.

The Volume Manager is very simplistic in this demo, implimented in **fsys.cpp**. It manages the registring and unregistring of file systems, and file system abstraction. You can call **volOpenFile()** to open a file. It defaults to opening **a:file.txt** but it will also work if you call it to open any file on any directory.

Not all file systems support subdirectories. Because of this, we leave subdirectory support to the file system drivers. Instead, the volume manager only handles the drive letter part of a path name. For example, if you call **volOpenFile ("a:\folder\file.txt")**, volOpenFile will pass **"\folder\file.txt"** to the file system registered on device 'a'. The file system driver is responsible for parsing the directory path name and opening subdirectories and files.

In the case of our FAT12 minidriver, this special routine is **fsysFatOpen()** which is responsible for parsing the direcory path (like "\folder\folder\file.txt") and calls its other file system routines for parsing and reading files and directories.

Thats it :-) This is possibly our last chapter covering FAT12. Because of this I do plan for an update covering writing files and directories on disk a little later.

Conclusion

This was a fun chapter, huh? We are now able to load files from disk. I know, I know, "About time!" :) We are almost now ready to make the big leap into multitasking and executing programs. Before going multitasking, however, we should cover Loaders. A Loader is responsible for loading and executing a program, and mapping it into an address space. We also need to cover heap management and stack management in address spaces.

Because I plan to update the memory management chapter heavily, I might move heap and stack management in a chapter following the memory management chapter. In any case, I will be sure to keep you updated on changes.

This does, however, mean that it is almost time for us to dive into multitasking. Afterwords? User mode!

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Chapter 21 Home Chapter 23

Operating Systems Development Series

Operating Systems Development - User land

by Mike, 2010

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

Welcome!

In the last chapter we have looked at VFS's and loaded and displayed a text file. We can use this VFS to also load program files that can be executed. This includes drivers, program software, shared, runtime libraries, and more.

In this chapter we will take the jump into supporting user land software. We will also be taking a look at System API's and how they work.

Lets get started!

Protection Levels

The Rings of Assembly Language

Kernel Land

In Chapter 5 we have took a quick look at the concept of the rings used in assembly language. These rings represent different protection levels. These protection levels are a hardware detail; they are implemented by the hardware.

Software running in ring 0 have the most control. They can execute priveldge instructions which allow the software to be able to perform more actions: hardware PIO, MMIO, Processor hardware controls and tables (like CPU cache controls and MMRs, et al.)

The list of privileged instructions have been shown in Chapter 7 however will also be listed here for completness.

If any software running in a protection level greater then 0 attempts to execute the above instructions, the processor generates a **Protection fault (#PF)** exception.

Privileged Level Instructions	
Instruction	Description
LGDT	Loads an address of a GDT into GDTR
LLDT	Loads an address of a LDT into LDTR
LTR	Loads a Task Register into TR
MOV *Control Register*	Copy data and store in Control Registers
LMSW	Load a new Machine Status WORD
CLTS	Clear Task Switch Flag in Control Register CR0
MOV *Debug Register*	Copy data and store in debug registers
INVD	Invalidate Cache without writeback
INVLPG	Invalidate TLB Entry
WBINVD	Invalidate Cache with writeback
HLT	Halt Processor
RDMSR	Read Model Specific Registers (MSR)
WRMSR	Write Model Specific Registers (MSR)
RDPMC	Read Performance Monitoring Counter
RDTSC	Read time Stamp Counter

The kernel or executive of an operating system typically resides in ring 0. Because of this, **kernel land** or **kernel mode** is any software running in **ring 0**. Ring 0 is also known as **Supervisor Mode**.

All of the software that we have written in this series so far has been kernel mode software: kernel and minidrivers. Microkernels and hybrids typically employ a more advanced driver interfacing scheme then what we use in the series that allow proper driver installation and drivers running in user mode, completely separate from the kernel. It is even possible to have a part of the kernel in user mode; it all depends on your design.

When the system is first started, the system is running in supervisor mode to allow the BIOS and operating system to start up.

User Land

Software running in ring 1 through ring 3 have less control of the machine then software running in ring 0. This is for protection of the machine; if there is an error caused by the software running in rings 1 through 3, the processor notifies the system executive or kernel of the problem using a **general protection (#GP)** exception.

Most operating systems employ a 2 mode system, kernel mode and user mode. While the x86 family supports 4 protection modes, these operating systems only use 2 for easier portability across architectures.

The design of these operating systems is for kernel mode software to run in ring 0 while user land software run in ring 3. Rings 1 and 2 are not used. Driver software can either operate in ring 0 to access hardware devices, or ring 3 using the provided driver API or System API to communicate with the hardware devices.

Because user mode software can not access hardware devices directly, they must notify the operating system in order to complete system tasks. This includes displaying text, obtaining input from user, printing a document, etc. These functions are provided to the user mode software in the form of libraries and APIs. These libraries and APIs communicate with the System API.

System API ... you have seen this term before. We will look closer on System APIs in a little bit. For now, lets take a closer look at user mode!

Ring -1

Some recent processors have a special protection level that allows a **hypervisor** ring 0 access. This is sometimes known as "Ring -1".

Welcome to User Land

There are a few steps required to enter user mode. (Come on, you didnt think it would be easy :)) Its not that bad though.

Step 1: Global Descriptor Table

We will be needing to go back to the **Global Descriptor Table (GDT)** first. The GDT was that big ugly structure that we needed when setting up protected mode for the first time. Recall that the GDT contains a list of 8 byte entries that contains information for the processor. Lets take another look at the GDT entry bit format again: (I have **bolded** the important parts)

- **Bits 56-63:** Bits 24-32 of the base address
- **Bit 55:** Granularity
  - **0:** None
  - **1:** Limit gets multiplied by 4K
- **Bit 54:** Segment type
  - **0:** 16 bit
  - **1:** 32 bit
- **Bit 53:** Reserved-Should be zero
- **Bits 52:** Reserved for OS use
- **Bits 48-51:** Bits 16-19 of the segment limit
- **Bit 47:** Segment is in memory (Used with Virtual Memory)
- **Bits 45-46: Descriptor Privilege Level**
  - **0: (Ring 0) Highest**
  - **1: (Ring 1)**
  - **2: (Ring 2)**
  - **3: (Ring 3) Lowest**
- **Bit 44:** Descriptor Bit
  - **0:** System Descriptor
  - **1:** Code or Data Descriptor
- **Bits 41-43:** Descriptor Type
  - **Bit 43:** Executable segment
    - **0:** Data Segment
    - **1:** Code Segment
  - **Bit 42:** Expansion direction (Data segments), conforming (Code Segments)
  - **Bit 41:** Readable and Writable
    - **0:** Read only (Data Segments); Execute only (Code Segments)
    - **1:** Read and write (Data Segments); Read and Execute (Code Segments)
- **Bit 40:** Access bit (Used with Virtual Memory)
- **Bits 16-39:** Bits 0-23 of the Base Address
- **Bits 0-15:** Bits 0-15 of the Segment Limit

Yikes, okay ... The **Descriptor Privilege Level (DPL)** bits above represents the priveldge level used for that descriptor. So, by setting those bits to 3, we effectivley make the descriptor a user mode descriptor.

So the first step is to create two new descriptors in the GDT - one for user mode data and the other for user mode code. This is done by modifying **i86_gdt_initialize** to add two new GDT entries for user mode code and data. Lets do that now:

```
//! initialize gdt
int i86_gdt_initialize () {

	//! etc...

	//! set default user mode code descriptor
	gdt_set_descriptor (3,0,0xffffffff,
		I86_GDT_DESC_READWRITE|I86_GDT_DESC_EXEC_CODE|I86_GDT_DESC_CODEDATA|
		I86_GDT_DESC_MEMORY|I86_GDT_DESC_DPL,
			I86_GDT_GRAND_4K | I86_GDT_GRAND_32BIT | I86_GDT_GRAND_LIMITHI_MASK);

	//! set default user mode data descriptor
	gdt_set_descriptor (4,0,0xffffffff,
		I86_GDT_DESC_READWRITE|I86_GDT_DESC_CODEDATA|I86_GDT_DESC_MEMORY|
		I86_GDT_DESC_DPL,
```

```
                        I86_GDT_GRAND_4K | I86_GDT_GRAND_32BIT | I86_GDT_GRAND_LIMITHI_MASK);

        // etc...

        return 0;
}
```

The above code is the same as what we did when creating the other GDT entries, with one change. Notice the I86_GDT_DESC_DPL flag. This will set both DPL bits to 2 which makes them for user mode (ring 3). Please note that none of this is new; all of the above flags were written from an ealier chapter when we covered protected mode.

Thats all that is needed! Note that the user mode code descriptor is installed at index 3 in the GDT, while the user mode data descriptor is at index 4. Remember that segment registers contain the offset of the selector it uses. Because each GDT entry is 8 bytes in size, it would be: **code selector 0x18** (8*3) and **data selector 0x20** (8*4).

So in order to use one of these selectors, just copy one of the above segment selectors into the segment register that will be used.

DPL

The **Descriptor Protection Level (DPL)** is the protection level of a segment descriptor. For example, our kernels code and data segments DPL are 0 for ring 0 access.

RPL

The **Requested Protection Level (RPL)** allows software to override the CPL to select a new protection level. This is what allows software to request changes to other protection levels, such as ring 0 to ring 3. The RPL is stored in bits 0 and 1 of a descriptor selector.

Wait, *what*? Remember that a segment selector is just an offset into the GDT. So, for example, 0x8 bytes was the offset for our ring 0 code descriptor. 0x10 was the offset of our data selector. 0x8 and 0x10 are **segment selectors**. GDT entries are all 8 bytes, so the value of a segment selector will always be a multiple of 8: 8, 16, 24, 32 etc. 8, in binary, is 1000. This means, with any value of a segment selector, the low three bits are zero.

The RPL is stored in the low two bits of the segment selector. So, if our segment selector is 0x8, the RPL is 0. If its 0xb (0x8 but with first two bits set, binary 1011 instead of 1000) the RPL is 3. This is required; this is how our software can switch to user mode.

CPL

The **Current Protection Level (CPL)** is the protection level of the currently executing program. The CPL is stored in bits 0 and 1 of SS and CS.

Remember that GDT entries are 8 bytes in size. Because segment registers, in protected mode, contain a segment selector (GDT entry offset), the low three bits are guaranteed to be zero. The low two bits of CS and SS are used to store the CPL of the software.

Protection Levels

If a software attempts to load a new segment into a segment register, the processor performs checks against the CPL of the software and RPL of the segment that it is trying to load. If the RPL is higher then the CPL, the software can load the segment. If it is not, the processor will raise a **General Protection Fault (#GP)**.

It is important to understand how RPL works, it is required information used when switching to user mode.

Step 2: The switch

Now we can make the switch to user mode!

There are two ways of performing the jump: Using **SYSEXIT** instruction or with an **IRET**. Both of these methods have their advantages and disadvantages so lets take a closer look. We will be using IRET in the series for portability purposes.

SYSEXIT Instruction

This section is planned to be expanded on.

IRET / IRETD Instruction

A lot of operating systems may employ this method as it is more portable then using SYSEXIT. Larger operating systems might even support this as a back up method in the case SYSEXIT is not avialable.

Okay, so how can IRET help us perform the switch? Recall from Chapter 3 the different methods used when switching modes. IRET is a trap return instruction. When executing an IRET, we can adjust the stack frame so it returns to user mode code.

When IRETD is executed, it expects the stack to have the following:

- SS
- ESP
- EFLAGS
- CS
- EIP

IRETD causes the processor to jump to CS:EIP, which it obtains from the stack. It also sets the EFLAGS register with the value above from the stack. SS:ESP will be set to point to the SS and ESP values that was obtained from the stack.

These are automatically pushed on the stack when an **INT** instruction is executed. Because of this, in the normal case these values would remain untouched. However, we can modify these values to cause IRET to perform a mode switch.

Okay, so first is setting the segment selectors. Recall that the low two bits represent the RPL that we want. In our case, we want 3 for user mode. So lets do that now:

```
void enter_usermode () {

	_asm {

		cli
		mov ax, 0x23	; user mode data selector is 0x20 (GDT entry 3). Also sets RPL to 3
		mov ds, ax
		mov es, ax
		mov fs, ax
		mov gs, ax
```

Now we can perform the switch to user mode. This is done by building the stack frame for IRET and issuing the IRET:

```
		push 0x23		; SS, notice it uses same selector as above
		push esp		; ESP
		pushfd			; EFLAGS
		push 0x1b		; CS, user mode code selector is 0x18. With RPL 3 this is 0x1b
		lea eax, [a]		; EIP first
		push eax

		iretd
	a:
		add esp, 4 // fix stack
	}
}
```

Notice that the stack frame matches that of what was in the list above. The IRETD instruction will cause 0x1B:a to be called in the above code inside of ring 3.

There is a slight problem however. If you try to use the above routine, or switch to user mode in a different way in the kernel, it will cause a **Page Fault (PF)** exception. This is do to the pages for the kernel being mapped for kernel mode-access only. To fix this, we will either need to enter user mode a different way or map the kernel so user mode software can access it.

For now, we are just going to map the kernel so user mode software can access it. This involves updating the **vmmngr_initialize()** routine and setting the USER bit in the PTEs and PDEs.

In a more complex operating system, this approach would not be used. This approach only works if we map kernel pages so they can be accessed by user mode software, which is bad. A more recommended approach is to keep kernel pages mapped for kernel-only access, and have the loader component of your kernel to map user mode pages when loading a user program. A stack and heap allocator would then map a region for program stack and heap to user mode. This current method **shares** the kernel stack with user land; larger systems should not do this.

Entering v8086 Mode

These are the same steps involved when setting up v8086 mode. v8086 mode requires a user mode task in order to enter v86 mode. Thus, by doing the above, you can enter v86 mode as well. However, there is one slight modification needed.

Recall the format of the EFLAGS register. Bit 17 (VM) is the **v8086 Mode Control Flag**. Because we push a value for EFLAGS on the stack when performing an IRET, in order to enter v86 mode, just set bit 17 of EFLAGS before pushing it on the stack. This will cause IRET to set the VM bit in the EFLAGS register on return.

Thats all that is needed to enter v8086 mode.

Notes on design

The above method presents an easy way to get into user mode, but at a cost: in order for the above method to work, the kernel region must be mapped to allow ring 3 software access to kernel memory. Because of this, while running in ring 3, the software-while will have some limitations do to protected mode, will be able to call kernel routines directly or even trash kernel space.

A possible way to resolve the above issues is to keep kernel memory reserved for ring 0 software. The loader component of the kernel can then map the necessary ring 3 regions of memory for the process *while* loading the program.

This will be looked at further in the next chapter when we develop a loader for the OS.

Switching back to kernel land

Step 1: Setting up the TSS

The x86 architecture supports hardware assisted task switching. This means the architecture includes hardware defined structures that allow the processor to select between different tasks.

Most modern operating systems do not utilize the hardware task switching support for portability purposes. These operating systems typically employ software task switching methods.

Task State Segment (TSS)

The TSS structure is quite large:

```
#ifdef _MSC_VER
#pragma pack (push, 1)
#endif

struct tss_entry {
        uint32_t prevTss;
        uint32_t esp0;
        uint32_t ss0;
        uint32_t esp1;
        uint32_t ss1;
        uint32_t esp2;
        uint32_t ss2;
        uint32_t cr3;
        uint32_t eip;
        uint32_t eflags;
        uint32_t eax;
        uint32_t ecx;
        uint32_t edx;
        uint32_t ebx;
        uint32_t esp;
        uint32_t ebp;
        uint32_t esi;
        uint32_t edi;
        uint32_t es;
        uint32_t cs;
        uint32_t ss;
        uint32_t ds;
        uint32_t fs;
        uint32_t gs;
        uint32_t ldt;
        uint16_t trap;
        uint16_t iomap;
};

#ifdef _MSC_VER
#pragma pack (pop, 1)
#endif
```

The TSS is used to store information about the state of the machine prior to a hardware task switch. It has a lot of members, so lets take a look!

- **General Fields:**
  - State of LDT,EIP,EFLAGS,CS,DS,ES,FS,GS,SS,EAX,EBX,ECX,EDX,ESP,EBP,ESI,EDI **prior** to task switch
- **prevTSS** - Segment Selector of previous TSS in task list
- **cr3** - PDBR, address of page directory for current task
- **trap**
  - **Bit 0:** 0: Disabled; 1: Raise Debug exception when task switch to task occurs
- **iomap** - 16 bit offset from TSS base to I/O permissions and interrupt redirection bit maps
- **esp0,esp1,esp2** - ESP stack pointers for ring 0, 1, and 2
- **ss0,ss1,ss2** - SS stack segments for ring 0, 1, and 2

Most of these fields are pretty simple. While we are not using hardware assisted task switching, we need to let the processor know how to go back to ring 0. Because of this, we need to set some of the fields in this structure - in particular the ring 0 stack and selector fields.

Step 2: Installing the TSS

Descriptor Segment

A TSS as implied by its name is a **segment**. Simular to all segments, the TSS requires an entry in the GDT. This allows us to control the TSS: setting if the task is busy or inactive; what software can access it (DPL) and other flags that can be set with descriptors. **The Base Address fields** must be the base address of the TSS structure that we set up.

LTR Instruction

The **LTR** (Load Task Register) instruction is used to load the TSS into **TSR** register. For example:

```
void flush_tss (uint16_t sel) {

        _asm ltr [sel]
}
```

ax is the segment selector for the TSS. Because the architecture supports hardware task switching, TSR stores the address of the TSS that defines the current task.

The **Task State Register (TSR)** is a register that stores the **TSS Selector**, **TSS Base Address** and **TSS Limit**. Only the TSS Selector can be modified by software however.

Installing the TSS

In order to install the TSS structure, first install the GDT entry for the TSS. Then select the TSS as the current task by calling **flush_tss** above.

```
void install_tss (uint32_t idx, uint16_t kernelSS, uint16_t kernelESP) {

        //! install TSS descriptor
        uint32_t base = (uint32_t) &TSS;
        gdt_set_descriptor (idx, base, base + sizeof (tss_entry),
                I86_GDT_DESC_ACCESS|I86_GDT_DESC_EXEC_CODE|I86_GDT_DESC_DPL|I86_GDT_DESC_MEMORY,
                0);

        //! initialize TSS
        memset ((void*) &TSS, 0, sizeof (tss_entry));

        TSS.ss0 = kernelSS;
        TSS.esp0 = kernelESP;

        TSS.cs=0x0b;
        TSS.ss = 0x13;
        TSS.es = 0x13;
        TSS.ds = 0x13;
        TSS.fs = 0x13;
        TSS.gs = 0x13;

        //! flush tss
        flush_tss (idx * sizeof (gdt_descriptor));
}
```

In the above code, **TSS** is a global structure definition for our **tss_entry** structure. We set up the TSSs selector entries to match the previous task (user mode selectors) and ring 0 stack (kernel stack, located at kernelSS:kernelESP). **flush_tss** installs the TSS into TSR.

Additional Instructions

There are a few other instructions that can be useful. All of these instructions can be executed by user mode software.

VERR Instruction

VERR (Verify Segment is Readable) can be used to check if a segment is readable. The processor will set the zero flag (ZF) to 1 if it can be read. This instruction can be executed at any proviledge level.

```
verr [ebx]
jz .readable
```

VERW Instruction

VERW (Verify Segment is Writable) can be used to check if a segment is writable. The processor will set the zero flag (ZF) to 1 if it can be written. This instruction can be executed at any proviledge level.

```
verw [ebx]
jz .readable
```

LSL Instruction

This instruction can be used to load the segment limit of a selector into a register.

```
lsl ebx, esp
jz .success
```

ARPL Instruction

This instruction can be used to adjust the RPL of a selector. It takes the form **arpl dest,src**, where dest is a memory location or register, src is a register. If the RPL of dest are less then src, the RPL bits of dest are set to the RPL bits of src. For example:

```
arpl ebx, esp
```

System API

Abstract

A **System API** provides tools, documentations, and interfaces that allow software to interact with the operating system. Different operating systems may use different termonology but the basic idea is the same. For example, Windows calls this API the "Native API".

The System API facilitates software interacting with the operating system and device drivers. The System API is the interface between user mode software and kernel mode software. Whenever the software needs system information or to perform a system task, such as creating a file, the software would invoke a system call.

A **System Call** also known as a **System Service** is a service provided by the operating system. This service is usually a function or routine. Software can invoke system calls in order to perform system tasks.

Design

SYSENTER / SYSEXIT

This section is planned to be expanded on.

Software Interrupt

Most System APIs are implemented by using a software interrupt. Software can use an instruction, like **int 0x21** to call an operating system service. For example, to call the DOS's Terminate function we would do:

```
mov ax, 0x4c00 ; function 0x4c (terminate) return code 0
int 0x21 ; call DOS service
```

In the above code, AH contains a function number. The int 0x21 calls the 0x21 interrupt vector to call DOS.

In order for the above to work, the operating system will need to install an ISR for interrupt vector 0x21. The ISR would be a **Finity State Machine (FSM)** that compares AH and passes control to the correct kernel mode function. And that, dear readers, is the design.

Software interrupts are more portable then SYSENTER and SYSEXIT. Because of this, most operating systems provide support for this method (possibly along with other methods.) We will be using this method in the series.

Examples

System APIs typically consist of hundreds of system calls.

This lists some operating systems and what methods they support. The INT numbers are software interrupt vector numbers using the above method.

- DOS: INT 0x21
- Win9x (95,98): INT 0x2F
- WinNT (2k,XP,Vista,7): INT 0x2E, SYSENTER/SYSEXIT, SYSCALL/SYSRET
  - Over 211 functions
- Linux: INT 0x80, SYSENTER/SYSEXIT
  - Over 190 functions

Basic System API

Step 1: System Call Table

Most System APIs implement a System Call table that contains all services. This table can be static, dynamic, auto generated, or a combination of the three. Large operating systems typically employ an auto-generated dynamic size table of system calls. This is do to the large number of system services that might be in this table; it would be very tedius to create it by hand.

For our purposes, we can just define a system service table in the kernel. It would contain the addresses of different functions that we have in the kernel that would like to be callable:

```
#define MAX_SYSCALL 3

void* _syscalls[] = {

        DebugPrintf
};
```

Hm, this table is quite small. We will add more to this list in the upcoming chapters, however it wont be too complex.

Because DebugPrintf is accessable from user mode (do to the kernel pages being mapped to allow this), and DebugPrintf not using any privedge instruction, the user mode software can technically call this routine directly without any problems. Depending on the design of your operating system or executive software this can cause security and stabability issues.

This is why it is typically recommended to keep the kernel pages accessable only from kernel mode. While it adds complexity to the software, the end result might be worth the effort.

Step 2: The Service Dispatcher

The next step is to create the service dispatcher ISR. Before that, we need to decide on what ISR to use... hm... Ill just follow Linux here and use 0x80. You can use any interrupt vector you like however, a lot of OSs use different vectors. So, lets install the ISR.

Remember that ISRs are stored in the IDT managed by the HAL layer. Also recall from chapter 15 that each IDT descriptor has its own DPL setting. **If the DPL of an IDT entery is less then the CPL, a GPF will result.** In other words, when we enter user mode, we can only call ISRs with IDT descriptors with DPL 3. Because we want our system interrupt callable from ring 3 software, we must install this ISR with the correct flags.

However, do to the current design of the HAL subsystem, this cannot be done by just calling setvect(), as this function does not allow us to set specific flags. To work around this issue, **setvect() has been modified with a second paramater to allow optional flags to be set.** This uses the C++ default paramater feature to achieve this so no other code needs to be updated.

```
void syscall_init () {

        //! install interrupt handler!
        setvect (0x80, syscall_dispatcher, I86_IDT_DESC_RING3);
}
```

Thats all there is to it :)

syscall_dispatcher is our ISR for system calls. This ISR will need to determin what system service to call by looking up the function in **_syscalls**. Usually System APIs use EAX to idenitify function numbers. We are going to do the same here. Thanks to the system service table we defined above, we can use EAX as an index. So, the function to call will be **_syscalls [eax]**.

```
_declspec(naked)
void syscall_dispatcher () {

        static int idx=0;
        _asm mov [idx], eax

        //! bounds check
        if (idx>=MAX_SYSCALL)
                _asm iretd

        //! get service
        static void* fnct = _syscalls[idx];
```

Okay, so now we have a pointer to the function to call. However now we a small problem. The above will effectively get a pointer to the service function we want based on the value given by EAX. However we dont know what function it is. We also dont know what to pass to the function nor the amount of paramaters it has.

One possible solution is to push all registers on the stack for the function call. Because the services are all C routines, we have to pass the paramaters in the way C functions would expect them.

```
        //! and call service
        _asm {
                push edi
                push esi
                push edx
                push ecx
                push ebx
                call fnct
                add esp, 20
                iretd
        }
}
```

Thats it :) The **add esp, 20** pops the 20 bytes off the stack that we pushed; and notice we return from the ISR with an **IRETD** instruction.

After the system software or executive installs their ISR to their respective interrupt vector, the software can call it by issuing a software interrupt. For example, if we call **syscall_init** to install our ISR, we can call a system service like this:

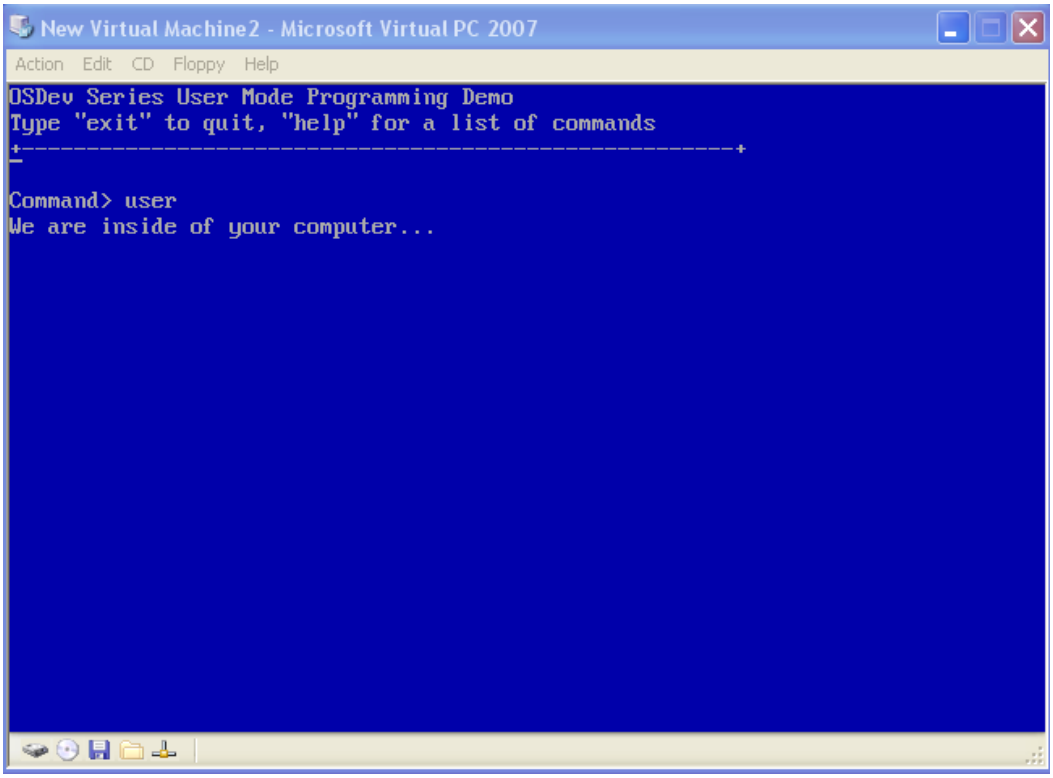
```
_asm {
        xor eax, eax ; function 0, DebugPrintf
        lea ebx, [stringToPrint]
        int 0x80 ; call OS
}
```

Notes on design

Most operating systems abstract the interrupt vector number and register details behind C interfaces. While calling system services in larger OSs directly is still possible, it is recommended to develop a standard C interface around the system services your system provides to user land software.

Large OSs typically wont have a system service for printing a message to the display. Rather it would contain services that can be called from user land software that allows user APIs to interact with kernel mode services, servers, or device drivers. Because of this, large OSs typically contain system APIs consisting of several hundred function calls.

Demo



Enter into user mode
Demo download

New and Modified Files

This chapter adds a few more files to the series demo. This includes:

- hal/tss.h
- hal/tss.cpp
  - Includes TSS routines described in this chapter
- hal/user.cpp
  - Includes user mode switching routines

This chapter also modifies the following files:

- kernel/mmngr_virtual.cpp
  - vmmngr_initialize has been updated to allow kernel pages to be user accessable
- hal/hal.h/cpp
  - set_vect() has been modified with an added second paramater
- hal/gdt.h
  - MAX_DESCRIPTORS has been redefined to 6 for the added GDT entries
- hal/gdt.cpp
  - Upgraded to include the installation of the user mode descriptors
- kernel/main.cpp
  - Updated to reflect new changes

Conclusion

Welcome to user land!

Now we have everything that is needed to switch between user land and kernel land. With this, we now have the capability of mapping user mode pages, loading, and running a program in usermode. We dont quite have the capability of returning back to the kernel of the OS in a nice way do to the system not managing tasks. We will look at this in the next chapter.

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Operating System Development Series

This series is intended to demonstrate and teach operating system development from the ground up.

24: Process Management

"If debugging is the process of removing software bugs, then programming must be the process of putting them in."
- Edsger Dijkstra

by Mike, 2012, 2013

Introduction

This chapter details the topics of process management and multitasking. The previous chapters have built a basic monolithic operating system capable of now supporting tasks. A lot of decisions made up to this point in developing the OS was to sacrifice complexity for simplicity; while still providing some detail in the complex designs. I will continue this trend in this chapter; the demonstration presented at the end by no means represents the only way to develop an OS. This chapter will cover the following topics:

1. Processes
2. Threads and tasks
3. Looking inside a process
4. Process management
5. Scheduling
6. Linking

Process management

Process management is the process by which operating systems manage processes, threads, enable processes to share information, protect process resources and allocate system resources to processes that request them in a safe manner. This can be a daunting task to the operating system developer and can be very complex in design. Lets take a closer look at each one of these.

Processes and threads

We will primarily be discussing processes and threads throughout the rest of this chapter. Process creation implies loading an executable image and creating at least one execution path (thread) to run it.

Inter-Process Communication

Inter-Process Communication (IPC) is a technique employed by many operating systems to allow communication between processes. This is typically done by message passing: the process would request to send a message to another process to the operating system which would send and queue the message to the other process if it is able to. IPC can be implemented in a number of different ways, the most common being files, pipes, sockets, message passing, signals, semaphores, shared memory, and memory mapped files. Operating systems may implement any or all of these methods of IPC. IPC is heavily used in some hybrid and monolithic kernel designs but is arguably most prominent in micro kernel designs.

This chapter primarily focuses on that of process and thread creation and thus will not focus on that of IPC. We may discuss IPC a little later but probably as an addition to this chapter.

Process protection

Loading multiple processes into the same address space poses a fundamental problem: both processes can read or write each other. The simple solution to this is loading the processes into their own virtual address spaces and mapping them to separate locations in the physical address space. The process can request to create more threads: this is on a per-process bases so all threads share the same address space as the process as you will soon see.

Process protection is also employed by mapping processes with the least amount of control that they need: for example, processes that must be in kernel land should be in kernel space, processes that do not need to be in kernel land should be in user space.

We will be utilizing both of these in this chapter when creating processes. They will be mapped into user space and in their own virtual address space. This means the process will not be able to access kernel pages (so cannot trash kernel stack or structures) nor can the process trash another process as they are in separate address spaces.

Resource allocation

Resource allocation means the safe manner of handing system resources (such as files or device handles) to processes that request them. Due to the early state of the series OS, there really is no resources that we need to worry about at this time. As things are needed, though, process resources that have been allocated are typically stored within the **process control block**. If you are wondering why we need to manage system resource allocation, consider the case in a multi-tasking environment when two processes try to open the same file at the same time and write to it.

With that in mind, we will be focusing on the creation of processes and threads in the following sections. We begin by providing a clear definition of what these are and what constitutes a process.

Processes

A **process** is an instance of a program, or part of a program, in memory. Processes are executed by the operating system or executive in order to perform complex tasks: such as play a movie or video, play a game, or even run the editor used to write this text in. In essence, one could say a process is a program – but a program could contain multiple processes. For example, a basic program to display a string might be built in its own program file. Loading the program might yield the operating system or executive to load other program files – **dynamic loading** of **shared libraries** containing executable code for the process to call and use. All of these program files are a part of the same process; its why a process can have an instance of multiple program files or even multiple instances.

Processes may execute in emulated or hardware environments by a **central processing unit (CPU)** or multiple CPUs or CPU cores. CPUs that support **hyper threading** or **parallel pipeling** can also execute multiple instructions from different processes at the same time. This means that the process may not be executed in a sequential manner (one instruction at a time) but may be executed in a number of different ways depending on the environment and hardware configuration. The IA32 CPU family defaults with these features disabled. This means that at computer startup, the CPU will execute all instructions one at a time (although might cache them in an **instruction cache buffer**). However if the operating system or executive were to enable these features for a process, the process and system must be designed in a way to be **multi-processor safe**. This one is an advanced topic however and is very easy to run into errors and thus will be discussed in an advanced chapter. We can break some processes apart into **threads** and **tasks**. We will look at these next.

Threads and tasks

Threads can be defined as a **single execution path inside of a process.** For example, in the most basic of an example we can have a program that just displays a message and returns:

```
#include <stdio.h>
int main (int argc, char** argv) {
   printf ("Hello, world!");
   return 0;
}
```

In this example, the process has a single thread: it begins at main() and the thread ends when the process terminates. (Do note however that this might not actually be the case: The runtime library that calls main() might contain threads.) Lets look at a multithreaded example:

```
#include <stdio.h>
static int _notExit = 0;

int thread (void* data) {
   while (_notExit) {
      /* do something useful */
   }
   return 0; /* thread terminates (returns to runtime which calls TerminateThread */
}

int main (int argc, char** argv) {
   CreateThread (thread);
   printf ("Hello, world!");
   return 0;
}
```

In this example, **CreateThread** calls the operating system to set **thread()** as a new **flow of execution**. After **CreateThread()** if called, **thread()** will be called by the operating system or executive and **execution will continue inside of both thread() and main() simultaneously** until one of them terminates. Notice that all processes are threads, but threads are not processes. A process can contain a single thread or many threads. Threads inside of a process can access and share the same global variables; although some compilers support **thread-local variables** as well.

Operating systems that support threads are said to be **multi-threading capable**. Examples of such operating systems include Windows, Linux, and Mac OS. **Tasks** are synonymous with threads. Tasks represent a task for the operating system or executive to execute. Thus any operating system that supports **multithreading** effectively supports **multitasking**. It is important to note though that not all multitasking operating systems supporting multi-threading.

What exactly is a process? We have currently defined a process as an instance of a program or a part of a program; lets elaborate on this definition by taking a deeper look inside of a process.

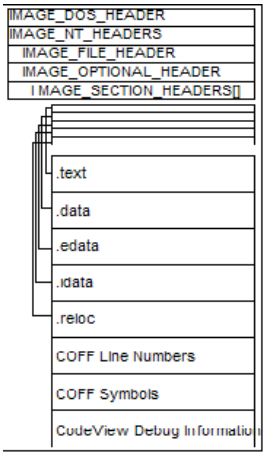
Looking inside a process

If we were to break apart a process, all we will see at the most fundamental level is code and data. Given ones' experience in programming, this makes sense: All programs are simply instructions instructing the CPU to perform an action or operating on data. It is understandable why programmers tend to separate the "data" part of the program from the "code" part in order to facilitate program development. **.data** and **.text** (for program code) are two of what later became many different types of **sections** within a program binary. Sections not only help program development but also facilitates standards for how different types of things can be stored within program binaries.

We will first discuss program sections and how they can be placed within a **process address space**. We can then discuss **symbolic information**, **debugging information, export and import tables,** and how they can be used.

Program sections

We have already looked at two sections (or **segments**): **.data** and **.text**. The program file contains these and other sections. The operating system or executive can load each section in the address space for the process to execute correctly. **Sections can also be relocated** during load time. This allows the operating system or executive to find the best location for each section if it needs to and update the process accordingly. However, **different program file formats support different things** so not all program file formats support section relocating.



The **Portable Executable (PE)** file format is the primary program file format used by the Windows operating system. The PE file format supports many different sections for code, data, resource data, symbolic information, manifest data, and more. Each section is stored within the binary file written by the linker or compiler. In order to see how it is stored in the file, we need to take a look at the format of the PE file format.

The above is an image of what is inside of the PE executable file. We can load an image into memory and parse the file contents as any other type of binary file so long as the sections do not need to be relocated. This is how the series boot loader is able to load our kernel image: The kernel in the series does not require section relocation so the boot loader can just load the file, locate the entry point in the headers, and call it directly. This is easy to do:

```
/* Get entry point from PE headers */
IMAGE_DOS_HEADER dosHeader = (IMAGE_DOS_HEADER*) imageBase;
IMAGE_NT_HEADERS ntHeaders = dosHeader->e_lfanew;
IMAGE_OPTIONAL_HEADER optHeader = &ntHeaders->OptionalHeader;
void (*EntryPoint) (void) = (void (*EntryPoint) (void) ) optHeader->AddressOfEntryPoint + optHeader->ImageBase;

/* Call program entry point */
EntryPoint ();
```

In a similar manner, we could parse the other headers and extract any type of information we want from the file: section information, debugging information, symbolic information, and more. **Kernel debuggers** and **user mode debuggers** typically use symbolic and debugging information to facilitate debugging. In other words, you could build your PE image with debugging information (or without); if you build it with you can attach a debugger to it remotely to perform source level debugging.

Having different sections inside of the executable files makes it easier for parsing and writing executable files. They provide a consistent location and way to store the data and reference them from headers. For example, in PE files there is a **.rscs** section that stores the actual resources (string table, bitmap images, program info, cursor, etc.) In order to locate a resource, we just need to parse the directory entry for it at **OptionalHeader->DirectoryEnteries [IMAGE_DIRECTORY_ENTRY_RESOURCE]** which gives us an RVA (relative pointer) to the resource tree structures that point to the resource data inside of the .rscs. section. The point is that the executable format has a specific format defined by the *PE file specification*. It has a specific format so there is a standard way to get information from the file.

Many compilers such as GCC and CL (Microsoft's compiler) is also capable of **programmer-defined sections**. In other words, programmers can also define their own section names and put whatever they need into that section. Operating system kernels and executives typically define special sections for different purposes. For example, both Linux and Windows define a special **.INIT** section containing one time initialization code and data. When initialization completes, the operating system kernel can free the section and re-use it for other things.

Common sections

There is a set of section names and types that are common to different object files and even architectures. It is important to be able to recognize these sections and what they are used for. They are as follows:

1. .text
2. .data
3. .bss
4. .rodata

The **.text** section is the common section name given to the section that contains program code. This is also known as the **code segment**. This section might be read only on some systems in order to prevent writing to it; however this prevents self modifying code (which is typically not recommended anyways.)

The **.data** section, as its name suggests, contains **static** and **global** data used by the program. It is always writable.

The **.bss** section is a part of the .data section and is typically used for statically allocated data that are initialized to zero. The .bss section is always cleared by the operating system loader so all of the data in it is set to zero. The name ".bss", according to Wikipedia, initially stood for Block Started by Symbol in the United Aircraft Symbolic Assembly Program. The .bss section contains all null variables and so does not take any space in the object files.

The **.rodata** section contains read only statically allocated data. It is typically common in Linux and Unix environments.

Notice that there is no section for temporary data. Recall that temporary variables are stored on the stack and thus do not need to be stored in the program file.

Custom sections in Microsoft Visual C

Microsoft's compiler provides several pragma directives that programmers can use to control where to place data and code in specific sections and making custom sections. These are:

1. alloc_text
2. code_seg
3. const_seg
4. data_seg
5. bss_seg
6. init_seg
7. section

The program loader does not need to worry about any special sections that the program has, all it needs to concern itself with is loading them into memory. The program (and thus programmer) has the responsibility.

Here is an example of using alloc_text for adding a function to a special section.

```
error_t DECL mmInitialize (SystemBoot* mb) {
   return SUCCESS;
}
#pragma alloc_text ( ".init" , mmInitialize);
```

In the above example, **mmInitialize** will be added to the section **.init**. This is a useful tactic used by some operating system kernels and executives. For example, the operating system kernel or executive can add or initialization code and data to a special .init section. Once initialization completes, the operating system can free that section to get some of the memory back.

Symbolic information

Symbolic information is the **symbols** that programmers give as names of addresses. For example, when we call a function like **printf()** how can the compiler and linker know what to do? Lets take a closer look.

"**printf"** is a symbol for a function defined in a library. When we call "printf()", the compiler adds the symbol **printf** to a **symbol table** managed by the compiler during the build process. Notice that the **name of the function is a symbol.** In a similar way, **static and global variables are also symbols**. We can argue that any name we give to an address (like a **label** in assembly language) is a symbol. Thus a symbol has two things: A name and an address.

If we build without linking the library containing the code for printf, the compiler is unable to output the final executable because it will not be able to translate the entire code to machine language. In other words, like an assembler, it cannot do anything with code like the following unless it knows what the function is:

```
call _printf
```

An assembler cannot completely assemble this instruction unless it knows the address of the symbol **_printf**. The assembler can not know the address if it does not know anything about the symbol. To resolve this, we declare that the symbol is **external** and the assembler or compiler outputs an **object file** instead of an executable file. It partially translates the instruction to machine code but in a form like this:

```
0xe8 _printf
```

This allows us to use another program—a *linker*—to resolve these symbols. The linker looks at the export symbol tables in the different object files and libraries for the symbol "_printf". If it is found, the linker can get the address of the function code and update the machine code with that address to properly link and output a final executable program. If the symbol is not found, it is *unresolved* and the linker gives us the famous "unresolved external symbol" error.

Symbolic information in the executable image can be used by debuggers to display human-readable information (functions and variable) names but at the cost of a bigger program file size.

There are different ways to "store" additional information about a symbol in the symbol name itself. This differs based on the **build environment** and c**alling convention**. The standard C calling convention is CDECL which just prepends an underscore to all names. So, for example, if we call "printf()"; its CDECL symbolic name is "_printf". C++ symbolic names differ between compilers and store a lot more information then just a name (such as return data types and operand types, namespace, classes, template names, etc.) Due to this, C++ symbolic names are said to undergo **name-mangling**. For example, the function "`void h(void)`" in CL (Microsoft's compiler) translates to the symbolic name `?h@@YAXXZ`. I will not go into the details of the name mangling format here.

Notice something interesting here. C symbolic names do not store anything about return data types or operands, but C++ symbolic names due to name mangling do. This is understandable and presents one of the many differences between the languages: With C compilers, you can call functions with different operand types or number of operands without error (although perhaps a warning when it can be detected); C++ compilers do.

Export and import tables

Symbols in a program library or object file can be **exported** for use by other libraries or programs. **Exported symbols** just tell the compiler and linker to add the respective symbol to the **export table**. Program files and shared libraries (Windows DLL's) may export symbols for use by other programs or debuggers. In a similar way, program files can request to **import** a symbol for use. This is where we can complete our printf() example above.

The Microsoft C Runtime Library is a shared library loaded with the program file. The operating system or executive can tell what DLL's a program file needs to operate by looking at the program **import table**. By default, CL (Microsoft's compiler) links with the Microsoft C Runtime Library import static library that contains the import table so the symbols are added and the respective DLL is included in the table. The operating system or executive must load all of the shared library files the program requires into memory if not already done so and update the program files **Import Address Table (IAT)** with the addresses of the functions in these other DLLs. The Microsoft C Runtime Library DLL that gets loaded not only includes the code for _printf but also exports the symbol _printf, so the operating system links them during runtime. (We will discuss that in more detail later.)

Thus, when we call "printf()" from a program file, this calls a jump table which calls the updated IAT address which calls the function "_printf" in the C runtime library DLL.

So far we have covered processes, threads, tasks, and took a look into what program files are and how they work. The goal of this chapter is to be able to load, execute, and manage multiple processes and tasks. Lets look at that next.

Process Management

Process Management is the management of **processes** in a software system. We have defined a process earlier as a program or a part of a program in memory. To manage processes, then, means managing multiple instances of programs in memory in a collaborated environment. This is typically a requirement in modern operating systems and implemented in the kernel or executive. Operating systems that support a form of process management is considered to be a **multitasking operating system.**

Representation

In order to manage a process, an operating system designer needs to determine how to best represent a process given OS design criteria and required system resources. A process consists of the following:

- Image of the executable in memory (machine code and data);
- Memory in use by the process and its virtual address space
- Descriptors used to represent the processes
- Process state information (registers, stack, attributes, etc.)

The operating system is required to manage the processes and allocate system resources in a fair manner to the processes that request them. Lets look at each of these closer.

Image of executable in memory

Executable programs are stored as files on disk to facilitate program loading and managing. **To load a program, an operating system loader loads the file into memory**. The loader must also be able to understand the type of file (it must be an executable the operating system can work with) and possibly support features of these file types (like resources and debug information.)

The image of the executable in memory is the current representation of the machine code and data of the image and how it appears in memory at any given time. We use the term "image" here to represent a "snapshot" of whats in memory. For example, its like taking a camera looking at a big array of bytes and taking a photo. The array of bytes can be machine code, data, or neither – we don't know nor care. Only the program instructions know.

Some data in the program image might be useful though to other programs or even the operating system itself. This is data the program image itself does not usually use; for example, the program file can contain debugging information. A debugger can be then attached to the program and use that information.

In short, the operating system needs to be able to load the file from disk into memory somewhere in order to execute it. This can be like just loading the file into memory "as-is". The operating system or another program can then get any useful data from the program file that it may need.

Memory in use by the process and its Virtual Address Space

Processes typically have calls to dynamically allocate memory and use stack space just like the operating system does. **The operating system is required to allocate space for a process stack and heap memory for the process to use**. For example, the operating system typically allocates a default stack size to all processes. **The executable file for the process however can also tell the operating system to allocate a larger stack space if the process needs it**.

The process heap is different. While the stack is allocated by the operating system before executing a process, the heap is not. Instead, each process has its own heap allocator in user mode. This is implemented in the C Runtime Library (CRT) using the familiar interface of malloc, free, realloc, brk, and sbrk functions. Programs that are linked with the CRT can call these functions to allocate memory. Programs that are not linked with the CRT however must implement their own heap allocator or link with another library that does.

The CRT Runtime implements a user mode heap allocator (typically a free list). The C function malloc might call brk, which calls the OS using the System API. The C function brk calls the OS in order to allocate more virtual memory to expand the heap when needed.

In short, the user mode heap works like this: The program calls malloc, which might call brk, which calls the OS using the System API to allocate virtual memory for the heap. The malloc and free family of functions implement their own user mode heap allocator. They only call the OS to allocator or free memory from the virtual address space.

In **preemptive multitasking**, all processes have their own virtual address space. This means every process must have their own Page Directory and associated page tables. In order to manage process specific information, we use a **process control block (PCB)**. Lets look at that next.

Descriptors used to represent processes

A **Process Control Block (PCB)** is a data structure used to store the information about a process or task. The PCB contains information such as interrupt descriptor pointers, Page Directory Base Register (PDBR). Protection level, running time, process state, process flags, VM86 flag, priority, and Process ID (PID). PCBs may contain a lot more information – its really OS specific.

Operating systems may used a linked list of PCB's to manage processes. When creating a new process, the operating system needs to allocate a new virtual address space, load and map the image, and attach a new PCB structure to the list. The scheduler uses the PCB list in determining what process to execute and to store the current state.

Process State Information

Process state information includes the entire register state of a process, in-memory state, Input/Output request state of the process at a given time. The process state is stored in the PCB when switching tasks. This is done by the heart of a multitasking operating system: the scheduler. Additionally, the current running state of the process is used to control the execution of processes by the operating system.

In the most simple case, a state is either RUNNING or NOT RUNNING. With this model, a process just created as stored in a NOT RUNNING queue and only labeled as RUNNING when it is in execution. The process that is NOT RUNNING may still exist in memory but in a waiting state until either the RUNNING process terminates or is interrupted by a process dispatcher inside of the scheduler.

In a three-state process management model, processes may either be RUNNING, READY, or BLOCKED. When a RUNNING process requests access to something that requires the process to wait (such as an I/O request) the operating system may change the process from RUNNING to BLOCKED. When the request can be performed, the process may be moved to either RUNNING or READY states. Processes in the READY state just means the process is ready for execution by the process dispatcher. Processes that are RUNNING are already being executed.

The final model is a five state process management model. This model utilizes five states: SUSPEND BLOCKED, BLOCKED, SUSPEND READY, READY, and SUSPENDED.

Scheduling

The **scheduler** is the component of an operating system kernel or executive that is responsible for task switching and CPU usage allocation. Operating systems employ scheduling algorithms to determine what task to execute next. Common scheduling algorithms used include but are not limited to First-in First-out, Shortest remaining time, Fixed priority preemptive, round-robin, and a multilevel queue. Possibly the most common algorithm used by both Windows and Linux is a **multilevel feedback queue**.

Basic process management support

Now we can implement basic process management support. The goal is simplicity so we will not be implementing an advanced multilevel feedback system with vm86 task support, I/O resource allocations, etc but will focus on a simpler but still efficient scheduler.

In order to do this, lets look at the goals of what we must do in order to add support:

1. Load and parse an executable image into memory;
2. Manage a list of PCBs for processes;
3. Support user mode tasks;
4. Support multiple virtual address spaces
5. Allocate stack space for each process; default size can be 4k;
6. Select a scheduling algorithm and implement task switching

These are the goals in order to support multitasking. The processes will be user mode processes. Multitasking, however relies on both **process management** and **scheduling**. Due to this, we will focus on building the framework to support multitasking but will only allow one process with one thread for this chapter. This will be extended in the next chapter as we implement a scheduler.

Process Control Block

The PCB structure for our system will be simple:

```
#define PROCESS_STATE_SLEEP  0
#define PROCESS_STATE_ACTIVE 1

#define MAX_THREAD 5

typedef struct _process {
   int            id;
   int            priority;
   pdirectory*    pageDirectory;
   int            state;
/* typedef struct _process* next; */
/* thread* threadList; */
   thread  threads[MAX_THREAD];
   /*
     note: we can add more information, such as the following:
       -LDT descriptor [if used]
       -Processor count being used
       -User and kernel times
       -Execution options, etc
   */
}process;
```

We can add more to this structure, but the above is really all we need. Notice that it stores the process ID (PID), priority and virtual address space. The two commented entries are provided for completeness only; in a typical OS they should be linked lists of processes and threads. This, however, requires a kernel heap allocator which we have not written. For simplicity, we will store 5 thread objects in the process as an array.

The final thing we need is a way to handle threads. All processes have at most one thread which starts execution at the entry point.

```
typedef struct _thread {
   process*  parent;
   void*     initialStack;
   void*     stackLimit;
   void*     kernelStack;
   uint32_t  priority;
   int       state;
   trapFrame frame;
}thread;
```

The thread structure stores general information about a thread in a process. Note that the structure stores a pointer to the parent process and information about the thread stack, priority, state (if its running or not), and a trap frame. The trap frame stores the current register state of a running thread.

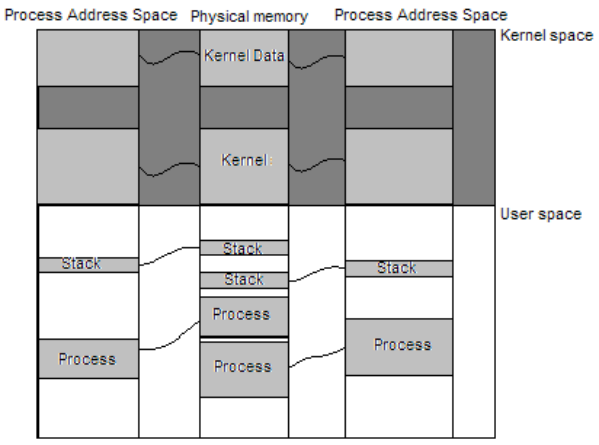
```
typedef struct _trapFrame {
   uint32_t esp;
   uint32_t ebp;
   uint32_t eip;
   uint32_t edi;
   uint32_t esi;
   uint32_t eax;
   uint32_t ebx;
   uint32_t ecx;
   uint32_t edx;
   uint32_t flags;
   /*
      note: we can add more registers to this.
      For a complete trap frame, you should add:
        -Debug registers
        -Segment registers
        -Error condition [if any]
        -v86 mode segment registers [if used]
   */
}trapFrame;
```

We will not be using the trap frame structure much in this chapter since we are not implementing multi-tasking yet. We will, however, be using the trap frame structure more in the next chapter as we develop a scheduler to store the current state of each thread.

Virtual Address spaces

A complication occurs when we want to support multiple virtual address spaces. **Each process address space consist of the entire 4GB address space where the kernel code and data is located at 2GB.** When we switch processes, we need to be able to switch address spaces—but only the low 2GBs of the address space ("user land"). In other words, lets say a user mode process is running. Somehow we need to be able to call the scheduler in the kernel to be able to switch tasks. Ah, but this means our kernel code needs to be in that same address space! If its not, its an instant crash.

To resolve this problem, we have to do just that—map our kernel code into every process address space. You might be wondering how this can be done. However it might become more clear when we consider that **multiple virtual addresses can refer to the same physical frame in memory**. In other words, we can map the kernel stack and code into both of the address spaces. Please see the following image.



The above image displays two virtual address spaces with the physical address space. Notice how the location of the process stack and code share different locations in physical memory. In other words, they are mapped to the same basic virtual address location to different physical address frames using our virtual memory manager. Lets consider the kernel for a moment. The kernel starts up in an environment with a single address space. It maps itself into its own address space during the initialization process. We need to still be able to map the kernel space into the other process address spaces as well to prevent problems. The kernel is already mapped into its own address space and is located at some place in physical memory. This means the kernel can re-map itself to the other process address spaces as well.

In the series kernel, the kernel maps itself from 1MB physical to 3GB virtual. The kernel must map 3GB region of all processes to 1MB physical, then, in order to map itself into each process address space. **The kernel and kernel stack must be mapped to the same location in every process address space**.

Operating systems can also map a portion of the kernel into the process address space rather then the entire kernel. This is quite common in large systems.

Address Space Management

We need to be able to work with being able to map virtual pages from different address spaces. More specifically, we need to be able to do the following:

1. Create a page table from any page directory
2. Map any physical address to virtual address from any page directory
3. Get the physical address of any virtual mapping from any page directory
4. Create new address spaces

The Virtual Memory Manager in the series currently does not support this functionality. We can quickly implement them, though, so lets do so now.

Creating a page table

To create a page table, all we need to do is to allocate a free frame (recall that a page table consist of 1024 PTEs which is 4096 bytes, the size of a page) and add it to the frame of a PDE in the page directory. V**irt >> 22** just allows us to get the directory index from the virtual address. If the PDE at pagedir [directory_index] is 0, then we know that this page table does not exist and so we allocate it using the physical memory manager. If it does exist, no need to allocate. We finish by clearing the page table which effectively sets its present bit to 0 (not present.)

```
int vmmngr_createPageTable (pdirectory* dir, uint32_t virt, uint32_t flags) {

        pd_entry* pagedir = dir->m_entries;
        if (pagedir [virt >> 22] == 0) {
         void* block = pmmngr_alloc_block();
         if (!block)
            return 0; /* Should call debugger */
          pagedir [virt >> 22] = ((uint32_t) block) | flags;
          memset ((uint32_t*) pagedir[virt >> 22], 0, 4096);

         /* map page table into directory */
         vmmngr_mapPhysicalAddress (dir, (uint32_t) block, (uint32_t) block, flags);
        }
         return 1; /* success */
}
```

This function allows us to create page tables for any page directory.

Mapping physical addresses

The next missing functionality is to be able to map physical to virtual addresses for different page directories. This one is easy.

```
void mapPhysicalAddress (pdirectory* dir, uint32_t virt, uint32_t phys, uint32_t flags) {

        pd_entry* pagedir = dir->m_entries;
        if (pagedir [virt >> 22] == 0)
                createPageTable (dir, virt, flags);
        ((uint32_t*) (pagedir[virt >> 22] & ~0xfff))[virt << 10 >> 10 >> 12] = phys | flags;
}
```

This function follows the basic functionality we implemented before in the virtual memory manager. We test for a valid page table, and create one if it is marked not present. The last line performs the mapping.

This function allows us to map physical to virtual addresses of any virtual address space.

Getting physical addresses

The next missing functionality is the reverse of what we did above: obtaining the physical address of any virtual address from a specific address space.

```
void* getPhysicalAddress (pdirectory* dir, uint32_t virt) {

        pd_entry* pagedir = dir->m_entries;
        if (pagedir [virt >> 22] == 0)
                return 0;
        return (void*) ((uint32_t*) (pagedir[virt >> 22] & ~0xfff))[virt << 10 >> 10 >> 12];
}
```

This function tests for a valid page table at that virtual address (by checking if its present) and returns the physical frame by dereferencing the PDE and PTE and returns the frame.

Creating a new address space

Each process runs in its own virtual address space. In order to achieve this, we must be able to create multiple address spaces.

```
pdirectory* createAddressSpace () {
        pdirectory* dir = 0;

        /* allocate page directory */
        dir = (pdirectory*) pmmngr_alloc_block ();
        if (!dir)
```

```
                return 0;

        /* clear memory (marks all page tables as not present) */
        memset (dir, 0, sizeof (pdirectory));
        return dir;
}
```

Notice the simplicity of this function: all it does is allocate a block and clears it. This makes sense as a page directory represents an address space and a page directory is 4096 bytes. By clearing, we are effectively setting the present bit to 0 in all the PDE's.

When it is time to execute a process we must be able to switch to this new address space that we just created. In other words, we need to be able to load this new page directory into the PDBR. We already implemented this functionality in the PMM. If we just load an empty page directory into the PDBR, however, we will surely triple fault right after. The cause of this is simple: none of our kernel code or stack is mapped into this new address space.

To resolve this, we just need to map kernel space. Interestingly, we can just copy the current page directory (stored in the PDBR) into this new address space like the following.

```
        memcpy (dst->m_entries, cur->m_entries, sizeof(uint32_t)*1024);
```

This is all that we need to do. We do not need to worry about copying any of the page tables as they were already mapped into the original page directory. The above effectively makes a copy of the address space – the kernel page tables mapped into both address spaces which is what we want.

Creating a thread

In order to create a thread, we need to first decide what our createThread function needs and what thread creation actually implies. Recall that we defined a thread as a single path of execution. Knowing this, all we need is an entry point function. When the function is completed, it calls the operating system to terminate the thread. This is typically done by the system API (such as the Win32 API) to simplify creating and terminating threads.

To create a thread, all that we need to do is allocate a thread structure and add it to the process. We were originally going to implement this functionality for this demo however decided to leave it for chapter 25 where we look at multi-threading.

Process creation

In order to create a process, we must already have a dedicated loader component for the operating system. The loader component is responsible for loading and parsing executable files, clearing the BSS section, section alignment, and any other thing you might want, such as dynamic loading of dynamic linked libraries. Creating a loader can be a complex task, specifically for a file format as complex as PE. Due to this, I opted to go for a simpler solution for the series so we can focus on the goal of this chapter: process management.

In order to crate a process, we must have a clear understanding of what a process is and how it differs from that of a thread. To be specific:

1. Threads each have their own dedicated stack; the process itself does not have one.
2. Each process must have at least one thread. This starts at the entry point of the process.
3. Each process must have their own virtual address space. Threads in a process share the same address space as the process.
4. Each process must be loaded from disk as an executable image. This is typically done using a separate loader component.

Due to the series not having a dedicated image loader component, for simplicity we will perform all of these steps in a single function called **createProcess**. The function follows the following steps.

1. Load the executable file.
2. Create the address space for the process.
3. Create a Process Control Block (PCB).
4. Create the main thread.
5. Map the image into the process virtual address space.

This is quite a lot for a single function. Remember, however, that it is better to separate the loader from process creation; later we can move the loading of the executable file to a dedicated loader. For simplicity, the routine assumes the same criteria of the boot loader. That is, the image to be loaded must have sector aligned sections (using the /align:512 flag) and can not be linked with any Microsoft Windows runtime libraries. While I might decide to add this functionality to the demo in the future, this complicates the loading code and—as noted earlier—is typically handled by the loader component anyways.

Process Address Space structure

At the moment, the kernel for the series OS has a lot of kernel structures loaded below the 1MB mark in identity mapped memory. This includes the kernel stack, initial page directory table and the page tables. The Direct Memory Access Controller (DMAC) memory region may also be located in this region. We must also take into consideration that the kernel also utilizes other memory regions (such as display memory) that are also in this identity mapped region.

This was all done for simplicity only. Typical kernels would initialize the kernel stack and initial page tables in kernel memory initially using Position Independent Code (PIC). PIC is also what allows higher half kernels to start when loaded at some other

physical base address. This is tricky to do right which is why I decided against it for the series. The result however created a mess: we now have a few kernel structures below 1MB.

Rather then moving things around, I decided that reserving 0-4MB for kernel mode only would be the best option. This allows the kernel to continue functioning with no modifications at all and no problems with remapping memory for display output and other basic things. In other words, the address space will look like this:

```
0x00000000-0x00400000 - Kernel reserved
0x00400000-0x80000000 - User land
0x80000000-0xffffffff - Kernel reserved
```

This means that all processes must have an image base within the region of 4 MB and 2 GB. I will be using 4MB as the base address of all user mode processes. The first 4 MB will remain identity mapped as kernel mode pages (this is already done); and the kernel itself will remain mapped at 3 GB. In short, a**ll pages for the process will be mapped as user mode pages between 4 MB and 2 GB.**

Creating a process

With all of that in mind, lets take a look at the function. This is a fairly long routine as it includes some software that is typically done in loaders. For this demo, the software loads and maps the image into the current address space rather then creating a new one; although both are implemented. This was done due to the software only designed to run one process at a time. We will change this in chapter 25 when we cover multitasking.

Notice two new functions - **vmmngr_createAddressSpace** and **mapKernelSpace**; these are not currently used in the demo. The first function allocates a new address space (we looked at this earlier in the chapter), the second maps kernel space into a virtual address space. That is, it maps the kernel memory, stack, page directory, and display memory into a new address space. Although these functions are not used, they will be used in the next chapter.

The **validateImage** function just parses the image headers and verifies that it is supported. Finally, although it creates an initial thread structure; it does not support multiple threads. It assumes one thread per process; where only one thread of one process can execute.

```
int createProcess (char* appname) {

        IMAGE_DOS_HEADER* dosHeader;
        IMAGE_NT_HEADERS* ntHeaders;
        FILE              file;
        pdirectory*       addressSpace;
        process*          proc;
        thread*           mainThread;
        unsigned char*    memory;
        uint32_t          i;
        unsigned char     buf[512];

       /* open file */
        file = volOpenFile (appname);
        if (file.flags == FS_INVALID)
                return 0;
        if (( file.flags & FS_DIRECTORY ) == FS_DIRECTORY)
                return 0;

        /* read 512 bytes into buffer */
        volReadFile ( &file, buf, 512);
        if (! validateImage (buf)) {
            volCloseFile ( &file );
            return 0;
        }
        dosHeader = (IMAGE_DOS_HEADER*)buf;
        ntHeaders = (IMAGE_NT_HEADERS*)(dosHeader->e_lfanew + (uint32_t)buf);

        /* get process virtual address space */
        //addressSpace = vmmngr_createAddressSpace ();
        addressSpace = vmmngr_get_directory ();
        if (!addressSpace) {
            volCloseFile (&file);
            return 0;
        }
        /*
           map kernel space into process address space.
           Only needed if creating new address space
        */
        //mapKernelSpace (addressSpace);

        /* create PCB */
        proc = getCurrentProcess();
        proc->id            = 1;
        proc->pageDirectory = addressSpace;
        proc->priority      = 1;
        proc->state         = PROCESS_STATE_ACTIVE;
        proc->threadCount   = 1;

        /* create thread descriptor */
        mainThread               = &proc->threads[0];
        mainThread->kernelStack  = 0;
        mainThread->parent       = proc;
        mainThread->priority     = 1;
        mainThread->state        = PROCESS_STATE_ACTIVE;
        mainThread->initialStack = 0;
        mainThread->stackLimit   = (void*) ((uint32_t) mainThread->initialStack + 4096);
        mainThread->imageBase    = ntHeaders->OptionalHeader.ImageBase;
        mainThread->imageSize    = ntHeaders->OptionalHeader.SizeOfImage;
        memset (&mainThread->frame, 0, sizeof (trapFrame));
        mainThread->frame.eip    = ntHeaders->OptionalHeader.AddressOfEntryPoint
                + ntHeaders->OptionalHeader.ImageBase;
        mainThread->frame.flags  = 0x200;

        /* copy our 512 block read above and rest of 4k block */
```

```
        memory = (unsigned char*)pmmngr_alloc_block();
        memset (memory, 0, 4096);
        memcpy (memory, buf, 512);

        /* load image into memory */
        for (i=1; i <= mainThread->imageSize/512; i++) {
           if (file.eof == 1)
              break;
           volReadFile ( &file, memory+512*i, 512);
        }

        /* map page into address space */
        vmmngr_mapPhysicalAddress (proc->pageDirectory,
                ntHeaders->OptionalHeader.ImageBase,
                (uint32_t) memory,
                I86_PTE_PRESENT|I86_PTE_WRITABLE|I86_PTE_USER);

        /* load and map rest of image */
        i = 1;
        while (file.eof != 1) {
                /* allocate new frame */
                unsigned char* cur = (unsigned char*)pmmngr_alloc_block();
                /* read block */
                int curBlock = 0;
                for (curBlock = 0; curBlock < 8; curBlock++) {
                        if (file.eof == 1)
                                break;
                        volReadFile ( &file, cur+512*curBlock, 512);
                }
                /* map page into process address space */
                vmmngr_mapPhysicalAddress (proc->pageDirectory,
                        ntHeaders->OptionalHeader.ImageBase + i*4096,
                        (uint32_t) cur,
                        I86_PTE_PRESENT|I86_PTE_WRITABLE|I86_PTE_USER);
                i++;
        }

         /* Create userspace stack (process esp=0x100000) */
         void* stack =
            (void*) (ntHeaders->OptionalHeader.ImageBase
             + ntHeaders->OptionalHeader.SizeOfImage + PAGE_SIZE);
         void* stackPhys = (void*) pmmngr_alloc_block ();

         /* map user process stack space */
         vmmngr_mapPhysicalAddress (addressSpace, (uint32_t) stack, (uint32_t) stackPhys,
                I86_PTE_PRESENT|I86_PTE_WRITABLE|I86_PTE_USER);

         /* final initialization */
         mainThread->initialStack = stack;
         mainThread->frame.esp    = (uint32_t)mainThread->initialStack;
         mainThread->frame.ebp    = mainThread->frame.esp;

         /* close file and return process ID */
         volCloseFile(&file);
         return proc->id;
}
```

Process execution

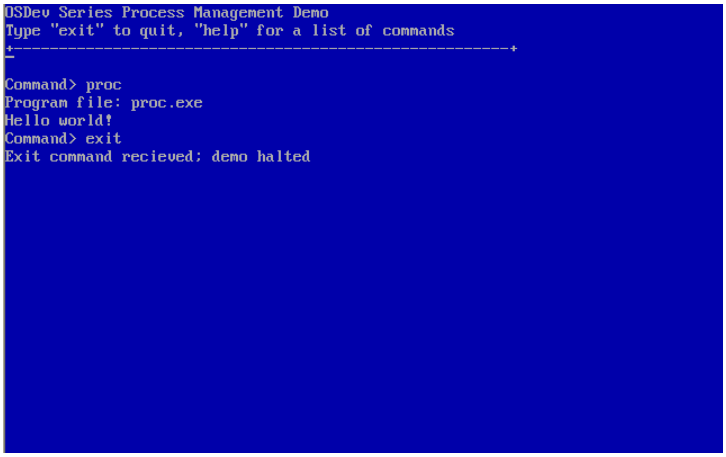
To execute a process, all we have to do is get EIP and ESP from the main thread in the process, drop to user mode, and execute it. We run into a problem though: how do we know what process to execute? Due to us having no scheduler yet, we can only execute one process at a time. This is done using a global process object that stores what process we are currently working with. **GetCurrentProcess()** returns a pointer to this object. We get ESP and EIP from its main thread, switch to the process address space, and drop to user mode to execute it.

Notice that we do **not** call **enter_usermode**. This is due to user mode software not being able to access kernel-only pages. If we called it, we will page fault. Instead, we just drop to user mode and execute the program directly using IRETD.

```
void executeProcess () {
        process* proc = 0;
        int entryPoint = 0;
        unsigned int procStack = 0;

        /* get running process */
        proc = getCurrentProcess();
                 if (proc->id==PROC_INVALID_ID)
                         return;
        if (!proc->pageDirectory)
                         return;

        /* get esp and eip of main thread */
        entryPoint = proc->threads[0].frame.eip;
        procStack  = proc->threads[0].frame.esp;

        /* switch to process address space */
        __asm cli
        pmmngr_load_PDBR ((physical_addr)proc->pageDirectory);

        /* execute process in user mode */
        __asm {
                mov     ax, 0x23          ; user mode data selector is 0x20 (GDT entry 3). Also sets RPL to 3
                mov     ds, ax
                mov     es, ax
                mov     fs, ax
                mov     gs, ax
                ;
```

```
                ; create stack frame
                ;
                push    0x23                    ; SS, notice it uses same selector as above
                push    [procStack]             ; stack
                push     0x200                  ; EFLAGS
                push     0x1b                   ; CS, user mode code selector is 0x18. With RPL 3 this is 0x1b
                push     [entryPoint]      ; EIP
                iretd
        }
}
```

Demo



Running a usermode process that uses the system API

Demo Download

This is an important milestone for any operating system in development; this milestone marks the beginning of interactivity and self hosting system designs. The demo uses the topics we looked at in this chapter, memory management chapter, and the PE loading chapter to implement a **proc (process)** command that loads an executable image (**Portable Executable** format), maps it into its own address space in user land, and interacts with the kernel using the **system API** through two system calls: **DebugPrintf** which can be used by the user land process to display strings using the kernel text terminal, and **TerminateProcess** which is used to terminate the process itself. The system calls are implemented using **software interrupts** that we looked at in earlier chapters.

Project "proc"

The usermode process we used is called **proc**. It is built as a 32 bit PE executable image, image base at 4 MB, with 512 byte section alignment. Here is the source for the project for reference.

```
void processEntry () {

        char* str="¥n¥rHello world!";

        __asm {

                /* display message through kernel terminal */
                mov ebx, str
                mov eax, 0
                int 0x80

                /* terminate */
                mov eax, 1
                int 0x80
        }
        for (;;);
}
```

Notice the process uses system calls to display the message and to terminate. These system calls are added to our system API implemented in earlier chapters. **Int 0x80 function 0 calls DebugPrintf** and **Int 0x80 function 1** is a new function, **TerminateProcess**. We can add more system services to improve the functionality of the demo; such as for file reading or input in a similar way.

TerminateProcess is responsible for cleaning up process resources and returning execution to the kernel command shell. Recall that when **int 0x80** is executed, the CPU traps into kernel mode and restores CS, SS, and ESP to their respective values from the TSS. Thus whenever any of the system calls execute, the CPU is in kernel land running in the same address space. This allows us to call kernel functions directly from **TerminateProcess** and to call the kernel command shell.

```
extern "C" {
void TerminateProcess () {
        process* cur = &_proc;
        if (cur->id==PROC_INVALID_ID)
                return;

        /* release threads */
        int i=0;
        thread* pThread = &cur->threads[i];

        /* get physical address of stack */
        void* stackFrame = vmmngr_getPhysicalAddress (cur->pageDirectory,
```

```
                (uint32_t) pThread->initialStack);

        /* unmap and release stack memory */
        vmmngr_unmapPhysicalAddress (cur->pageDirectory, (uint32_t) pThread->initialStack);
        pmmngr_free_block (stackFrame);

        /* unmap and release image memory */
        for (uint32_t page = 0; page < pThread->imageSize/PAGE_SIZE; page++) {
                uint32_t phys = 0;
                uint32_t virt = 0;

                /* get virtual address of page */
                virt = pThread->imageBase + (page * PAGE_SIZE);

                /* get physical address of page */
                phys = (uint32_t) vmmngr_getPhysicalAddress (cur->pageDirectory, virt);

                /* unmap and release page */
                vmmngr_unmapPhysicalAddress (cur->pageDirectory, virt);
                pmmngr_free_block ((void*)phys);
        }

        /* restore kernel selectors */
        __asm {
                cli
                mov eax, 0x10
                mov ds, ax
                mov es, ax
                mov fs, ax
                mov gs, ax
                sti
        }

        /* return to kernel command shell */
        run ();

        DebugPrintf ("¥nExit command recieved; demo halted");
        for (;;);
}
} // extern "C"
```

Bug report

There is a reoccurring bug that has been fixed in a few previous demo's but may still be present in others. We plan to upload the fix for all the demo's that it may be present in the future. The bug is in **vmmngr_initialize**, where some demo's call this function prior to initializing the physical memory manager and those demo's also improperly map kernel space thus may result in page fault or triple fault. It has been resolved (again) in this chapter's demo so please check **main.cpp** and **mmngr_virt.cpp** for the updated code.

Updated file list

- **sysapi.h** - **_syscalls** has been updated to include **DebugPrintf** and **TerminateProcess.**
- **task.h** – New.
- **task.cpp** – New.
- **main.cpp** – Added **proc** command. Also VMM bug fix.
- **mmngr_virt.h** – New address space functions.
- **mmngr_virt.cpp** – New address space functions. Also VMM bug fix.
- **image.h** – PE image structures and definitions.
- **proc/main.cpp** – New.

Conclusion

In this chapter we have looked at processes, threads, process management, and built basic process management support. We have covered everything needed for executing user mode programs from disk which marks a big milestone for the operating system.

In the next chapter we will build on the process management functionality implemented in this chapter to build a scheduler and complete preemptive multitasking support.

Until next time,

~Mike ();

OS Development Series Editor

Resources

The following links were referenced to provide more through and accurate information. Please reference them for additional information.

http://en.wikipedia.org/wiki/Process_(computing)

http://en.wikipedia.org/wiki/Scheduling_(computing)#Scheduling_disciplines

http://en.wikipedia.org/wiki/Process_management_(computing)

Additional links

The following links are additional tutorials or resources related to this topic. They might be helpful as a supplement to the material or even help with providing different designs. If you know of any additional links that might be helpful to add, please let me know. Links may also include some multi-tasking concepts that will not be looked at in depth until the next chapter.

http://www.jamesmolloy.co.uk/tutorial_html/9.-Multitasking.html

 Chapter 23

Home

Operating System Development Series

This series is intended to demonstrate and teach operating system development from the ground up.

25: Process Management 2

"Controlling complexity is the essence of computer programming" - Brian W. Kernighan

by Mike, 2015

1. Introduction

Welcome!

In the previous chapter we detailed basic process management topics, including Inter-Process Communication (IPC), protection, resource allocation, Process Control Block (PCB), process execution states, and process address spaces. We have also detailed single tasking support and implementing basic single tasking. This chapter is a continuation of the previous chapter and will go into more detail of the respective topics; with emphasis on multitasking, scheduling, security, and mutual exclusion. In particular, we will cover:

1. Multithreading;
2. Multitasking;
3. Init and Idle Process;
4. Kernel/User Shared Data Space;
5. Mutual exclusion and Semaphores;
6. Introduction to Concurrent programming;
7. Scheduling algorithms;
8. Introducing to the MP Standard.

We will assume that you have read the previous chapter and so this chapter will be more advanced; focusing on real world designs and implementations. Like the previous chapter, we will first dive into the theory behind these topics and then present a demo that will implement complete multi-threading in user land processes. Also note that we only provide a brief introduction to the MP standard; we may cover it in more detail in a later article. Implementing MP support requires proper support for the APIC which is an advanced topic.

2. Process State Management

We already talked a lot about processes throughout the series so this will just be a review of process states and process creation. In the previous chapter we implemented a function for creating a process. We will be modifying it in the accompanying demo to create a new task for the process so that it can be properly executed. We need to review state management since it ties closely with the scheduling of processes.

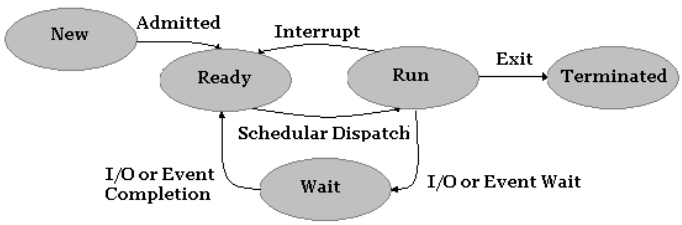
The **state** of the process is the current activity employed by the process. At a minimum, the process can be **created**, **executed**, **ready** to be executed, and **terminated**. Already this gives us four possible states:

- **New**. The process is being created.
- **Running**. The process is executing.
- **Ready**. The process is ready to be executed.
- **Terminated**. The process has completed.

This is a good start. However we can do better then this. Let's say that we have some process running, and the process sends a request to read a large file from the disk. However, in a system with multiple processes, the disk may be busy handling the request from that process. Our process needs to **Wait** until the **Input/Output Request** can be completed. For another example, let's say that we have two processes, but they communicate with each other through **signals**. A process would need to **Wait** for a signal to be **raised**. This would be our fifth state:

- **Wait**. The process is waiting to complete an I/O request, exception, or signal.

Putting everything together, a process goes through the following states.



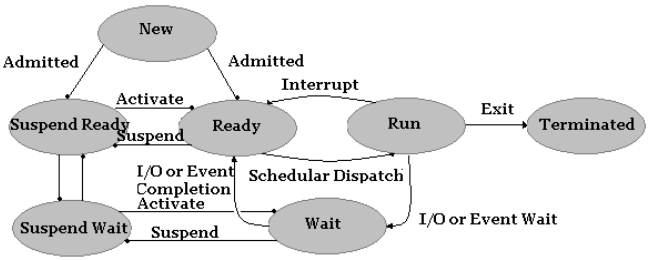
The above diagram illustrates the current state model. **New** processes are **admitted** into the system **Ready queue**. When the

Scheduler dispatcher selects the process to run, the process enters the **Run** state. From here, the process may take any number of state changes. If an **interrupt** or **exception** fires, the **Scheduler Dispatcher** may need to switch to another process which involves moving our process back into the **Ready queue**. If, instead our process tries to read from a file, the process will initiate the **I/O request** and be placed on the **Wait queue** until the request is completed. When the I/O request is satisfied, the process will be placed back into the **Ready queue** to be selected by the **Scheduler dispatcher** again. Finally, at any time while running the process terminates, it will be terminated.

Sometimes it might be helpful to **suspend** a process. This involves taking the process out of memory and storing its state on the disk. This is specifically useful when freeing up system resources and allows other processes with higher priority to run. This requires, at a minimum, two more states:

- **Suspend Ready**.
- **Suspend Wait**.

Adding these to our previous diagram, we have the following.



Processes in the **Ready** or **Wait** state may be **Suspended** depending on the resource demand of the system. There can be many more states that you can add to this depending on your design needs, however for most general purpose operating systems, the above state diagram would suffice.

For our purposes, we will only be concerned with the **Ready**, **Run**, and **Terminated** states. However we may also implement the **Wait** state to properly support the **sleep** function in the accompanying demo.

3. Concurrent Programming

So we looked at process states and state management and process creation. We have also dived deep into a process in memory with the previous chapter. The final topic that we need to dive into is ***multitasking***. The heart of multitasking is the **Scheduler dispatcher** which we will cover in the next section. The Scheduler dispatcher is responsible for moving processes between states and schedule processes for execution. This is why we covered those first – we will be using them in that section later on. Before moving on to the scheduler however, we need to take a closer look at what happens with multitasking when there are multiple **threads of execution**. When two threads or processes run ***concurrently*** and share some data with each other, it becomes critical to synchronize the activities between the two threads of execution.

Concurrency means that the current state of the process is not known. When multiple processes run along side each other and share data with each other, they are said to be running ***concurrently***. ***Concurrent programming*** defines the set of techniques used to **synchronize** access to **shared resources** between concurrent processes or threads.

Critical Section Problem

On single core systems, the operating system will allocate a small amount of time for execution for each process. The system switches rapidly between the different processes running concurrently. Processes may be interrupted at any time. In addition, systems that support **parallel execution** may execute instructions from different processes at the same time.

To see the problem of current programming, consider two processes with the following instructions.

Process A	Process B
`mov eax, [count]`	`mov ebx, [count]`
`inc eax`	`dec ebx`
`mov [count], eax`	`mov [count], ebx`

If we are to execute these processes concurrently, they would be ***interleaved*** in some order when the scheduler switches between the two processes. There are many different ways the processes may be interleaved, one way might be:

```
mov eax, [count]
inc eax
mov ebx, [count]
dec ebx
mov [count], eax
mov [count], ebx
```

If **count** is shared between the two different processes, you might notice a big problem here. Because there is no control over the **order of execution**, we cannot insure that the value of count is valid because we may get different results depending on *when* the scheduler decides to switch between the two processes. The outcome depends on who reads and writes the variable first. What we have is a **race condition**.

To combat the race condition, we need to **guard** the variable while it is being used by another process. We need to **synchronize** the two processes in some way. This is a part of the **critical section problem**.

The problem is *compounded* on systems with multiple processors since the current execution state and current instruction streams are interleaved while executing a *single* process.

The Problem.

We need a method to control synchronization of a process that executes concurrently. When a critical section request is made, we must insure that only *one* processor executes the code within the critical section until it completes. Farther, we must insure that other processes and threads do not execute while we enter the critical section.

The Criteria.

- **Mutual Exclusion.** When a process is executing in a critical section, no other process is executing in a critical section.
- **Progress**. Processes do not wait indefinitely to enter their critical section.
- **Bounded Waiting.** The amount of time between making the request to enter its critical section and actually entering it must be bounded.

Semaphores

Sow how do we implement **mutual exclusion**? We need some form of **cooperation** between the two processes. **If process A is operating on a shared resource, and process B needs access to it, we want process B to wait**. However, **once process A is done with the resource, we want it to signal that Process B can now use that resource**. Thus only one process can ever use the shared recourse at any given time. This is ***mutual exclusion***.

What we can do is introduce another variable to keep track of whether or not the resource is currently being used or not. This variable is called a ***lock***. We can then use this lock to keep track of the other resource.

- If the lock is 1, the resource is in use by some other process.
- If the lock is 0, the resource is free for use.

This type of lock has a special name. It is called a ***mutex***. The mutex has only two values and is also called a ***binary semaphore***. Recall what we need to do: we need one process to ***wait*** and the other process to ***signal***. These are the basic functions we will be using throughout this chapter.

```
atomic Wait (Semaphore S) {
while (S <= 0)
  Place process on S.Queue and block.
  S--;
}
atomic Signal (Semaphore S) {
  S++;
}
```

The **mutex** is just a ***binary semaphore*** with values of 0 or 1 only. ***Semaphores*** are generalized locks and are not restricted (that is, whereas the mutex only has two values, general semaphores do not.) Also notice the atomic keyword in the above code. This implies that the code will never be interrupted when it is executed. That is, it is ***guaranteed*** to run as a block of code on a single processor in the correct order. ***They are to be treated as a single unit*** (hence are called **atomic** operations.)

Unfortunately, it isn't quite as simple as what we shown above. **Atomic operations are hardware dependent** and so we need some assistance from the processor to make it work. More specifically, we need to make use of the LOCK instruction prefix. We will discuss this in more detail later as we implement these primitives into actual code.

For now, we believe that it is best to see some examples of using semaphores since they can be difficult when first introduced. It is important to get some practice with using them since you will be using them a lot if you ever plan to completely support multiprocessing.

Example. We opened up this section by showing how the instruction flow can get interleaved as we swap between different processes. The problem was that both of the processes can be executed at any time, and because they share a resource, there was no way to verify the integrity of the resource. We can fix that with semaphores. Assuming **count** is a global variable that is shared by two processes, we can use semaphores to synchronize access to it. Note that **signal** and **wait** are **atomic** operations.

Process A	Process B
count++; signal (s);	wait (s); count--; signal (s);

Spinlocks

Mutual exclusion is the first criteria for a solution to the critical section problem. This means that, when one process enters a critical section, no other processes can enter a critical section. To implement this functionality, we need some method of implementing an **atomic** operation that can guarantee mutual exclusion. One idea is to use a simple variable to act as a lock. If the lock is 1, some process is inside of the critical section. So the first idea is,

int lock=0;

Process A	Process B

while(1) {
  if (!lock)
  lock = 1;
  do_something();
  lock=0;
}
	while(1) {
	  if (!lock)
	  lock = 1;
	  do_something();
	  lock=0;
	}

Pretty simple. The lock starts at 0, so whatever process runs first will detect this and set the lock. When its done, it releases the lock so the second process can now use it. This would work somewhat, but there is still a big problem. Let's say that process A detects that the lock is 0 but gets interrupted by process B before process A has a chance to set the lock. So process B detects the lock is also 0 and now sets it. So if process B gets interrupted somewhere in **do_something**, process A will continue executing – as if the lock was still 0! And so both processes can still enter the same critical section (in this example, the critical section is the call to **do_something**) at the same time if the processes get interrupted when trying to read and lock the lock variable itself. This seems like a small error, but it can quickly propagate and will happen quite a lot.

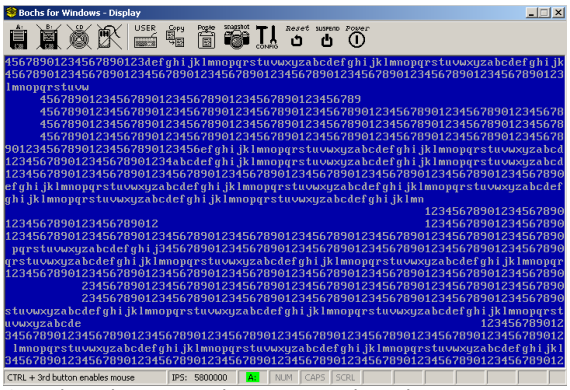
So the problem here is that we cannot guarantee that accessing and setting the lock can be done without getting interrupted. The operation is not ***atomic***.

To be able to visualize what can happen without actual atomic operations, let's say that we have two threads. The first thread displays characters a-z and the second thread displays numbers 0-9. They run concurrently using the scheduler that we develop later on. Here are the threads,

```
Process A
void task_1() {
  char c='a';
  while(1) {
    DebugPutc(c++);
    if (c>'z') c='a';
  }
}
```

```
Process B
void task_1() {
  char c='0';
  while(1) {
    DebugPutc(c++);
    if (c>'9') c='0';
  }
}
```

As these two tasks run concurrently, the output will become an interleaved mess. The reason is that both processes are reading and writing from shared resources without care. Even if we were to introduce a lock as we discussed above, the output wouldn't be much better. In this example, the shared resources are video memory and the global variables used by **DebugPutc** which is responsible for cursor positioning and scrolling. As one process reads the current x or y position or prepares to scroll, it may be interrupted and the position and other global variables can be mangled without the first process ever knowing.



Sample without semaphores. Notice how the output is a mess.

So to fix this, we need something more then a simple lock. Our direction is good – but we need hardware support. If there was a method to make it so that we can test and set a lock variable in one single operation with the guarantee that it will never be interrupted (so it is **atomic**) we can finally satisfy the **mutual exclusion** criteria.

One such hardware primitive is the **LOCK** instruction prefix. This prefix locks the system bus from reads and writes while the instruction is being performed. Because the data bus is locked, it is guaranteed to be atomic. So a simple **LOCK XCHG** or **LOCK BTS** can be used when setting and testing a lock variable. For example,

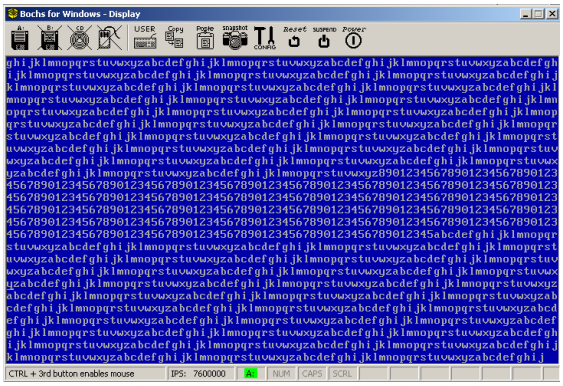
```
inline void acquire(int* lock) {
  _asm{
    mov eax,[lock]
a:  lock bts [eax], 0
    pause
    jc a
  }
}

inline void release(int* lock) {
  _asm{
    mov eax, [lock]
    mov [eax], 0
  }
}
```

We can now call these functions from to acquire and release the lock.

```
Process A
void task_1() {
  char c='a';
  while(1) {
    acquire(lock);
    DebugPutc(c++);
    release(lock);
    if (c>'z') c='a';
  }
}
```

```
Process B
void task_2() {
  char c='a';
  while(1) {
    acquire(lock);
    DebugPutc(c++);
    release(lock);
    if (c>'z') c='a';
  }
}
```

And we get the desired result.



Running Sample with spinlocks. Note how the display is now nicely in order.

4. Classic Concurrency Problems

Producer / Consumer Problem (Bounded Buffer Problem)

This is the first classic concurrency problem we will look at. Suppose that we have two independent processes, one called the **producer** and the other called the **consumer**. Let's also assume that there is a shared buffer being used by both of the processes. The producer is responsible for putting data into the buffer and the consumer is responsible for taking data out. This is the basic setup for the classic **Producer/Consumer Problem** also known as the **Bounded Buffer Problem**. The problem is that we need to make sure that the producer does not add data to the buffer if its already full and the consumer does not try to remove data from a buffer that is empty. The problem gets more interesting when there are multiple producers and consumers.

Example. This is a solution to the Bounded Buffer problem. This assumes a single producer and consumer running concurrently.

```
Semaphore c = 0;
Semaphore s = BUFFER_SIZE;
```

```
Producer                Consumer
while (true) {          while (true) {
   item = produce ();      wait(c);
   wait(s);                item = read();
   write(item);            signal(s);
   signal(c);              consume(item);
}                       }
```

Readers / Writers Problem

The classic R**eaders/Writers problem** is when an object is shared among many processes such that there are two types of processes – **readers** and **writers**. **Readers** read the shared data but never modify it. **Writers** can read data and modify it. **Many readers may read the data concurrently**.

Example. There are many different solutions and versions of this problem, this is one of them. Note that we use two semaphores here so that we can allow multiple readers at the same time.

```
Semaphore c = 1;
Semaphore s = 1;
int count = 0;
```

```
Writer          Reader
while (true) {  while(true) {
   wait(c);       wait(s);
   write();       count++;
   signal(c);     if (count == 0)
}                   wait(c);
                  signal(s);
                  read();
                  wait(s);
                  count--;
                  if (count == 0)
                    signal(c);
                  signal(s);
                }
```

5. Inter-Process Communication

Inter-Process Communication (IPC) is the technique supported by operating systems that permit processes to signal and share data with other running processes. There are many different types of techniques for implementing IPC protocols, we will introduce some of the most commonly used ones here.

Pipes

Pipes are a basic technique that uses a circular buffer to store the data between a producer and consumer. The producer writes data to the buffer and the consumer reads from it. There can be multiple producers and consumers of the data. There are two types of pipes, anonymous pipes and named pipes. **Named pipes** are given a name and appear as a file object in the virtual file system. Any process in the system can **open** named pipes. **Anonymous pipes** can only be opened by child processes that inherit it from the parent.

The operating system must provide functionality for **storing** the data stream that is shared between the consumers and producers, **reading** and **writing** the stream, and **blocking** processes that attempt to read from the pipe when there is no data to read. The operating system must **synchronize** the reading and writing using the mutual exclusion techniques that we discussed above. This is usually done using a **First-In-First-Out (FIFO) circular buffer** and using **semaphores** to synchronize access to it when reading and writing it.

Pipes are **file system objects**. When you **Open** a pipe, you will get a **File Descriptor** pointer back. So you can use the file **Read** and **Write** methods to read and write to the pipe as if it were a file. **Open file handles are inherited by child processes**, and so pipes are also inherited.

Pipes can be managed just like file system descriptors. The **Process Parameter Block** stores a pointer to a **Process Handle Table** that stored all open references to file descriptors, pipes, and other system objects. They can also be trivial to implement on systems that already support **device files**.

Message Passing

The basic idea is simple enough – a producer sends a **message** and a consumer takes it. There might be additional problems depending on if we want to support **synchronized** or **asynchronous** message passing. We also must think about how to **store** the messages, where they should be **managed**, the **format** of the messages, and how to verify that messages are delivered in the expected format to the expected process.

So, what exactly is a *message*? Messages are whatever the process wants it to be. The consumer and producer must agree on some type of **protocol** for how to interpret the message. They both need to know the **data structure** of the message. From the operating system side, the OS does not care about the format of the data – **unless its an OS defined message which is typical of microkernels**.

The operating system needs to implement support to **Send** and **Receive** messages at a minimum.

Synchronous Message Passing

For Synchronous message passing, we need at a minimum two functions. Assuming **J** and **K** are process identifiers (PID)'s,

- send(J, message)
- receive(K, &message)

The producer calls **send** to post a message. With synchronous message passing, the producer gets put in the **suspended queue** until **J** calls **receive** to get the message. When **J** calls **receive**, the operating system can copy the message sent to **J** directly and resume **J**. The operating system can then put the producer back on the **waiting queue** so that it can be executed by the scheduler. Synchronous message passing does not require a message queue since only one (the producer or consumer) will ever be running at the same time (the other would be **suspended** or **waiting**.)

Asynchronous Message Passing

Asynchronous message passing also needs a minimum of two functions,

- send(J, message)
- receive(K, &message)

The producer calls **send** to post a message and the consumer calls **receive** to obtain the message. With asynchronous message passing, the operating system maintains a **message queue** per process. Producers can **send** messages at any time and will not be suspended. Messages are copied to the end of the message queue. The consumer can then **receive** the message from the front of the message queue. The message queue itself is allocated in **kernel memory**; a dedicated pointer in the Process Control Block points to the queue.

We now have an interesting question. With **synchronous message passing**, when the process calls **receive** and there is no process that sent any message, the process gets suspended until another process calls **send**. With **asynchronous message passing**, we have two options:

- We can **suspend** the process that called **receive** or,
- We can have **receive** return a status code and just continue **running** the current process.

It turns out that the better approach is to offer a few more functions,

- send(process, message)
- receive(process, &message)
- sendrec(process, &message)
- notify(process, message)

Shared Memory

When we map the same physical frames into the virtual address spaces of two or more processes, it is **shared** among those processes. Both processes would be able to read or write to the same pages (depending on the **security attributes** set when mapping pages. (For example, you can map the physical frames as read/write for process A but as read-only for process B.) Operating systems typically provide support of shared memory through **memory mapped files**. Under Windows, for example, you would first call **CreateFile** or **OpenFile** on a named memory mapped file object followed by **MapViewOfFile** which maps the region of memory into the process address space and returns a pointer to it.

6. Scheduling

The scheduler is responsible for the allocation of system resources. System resources include the CPU, memory, and system devices. There are typically many schedulers, however they tend to fall under three categories: **short term**, **medium term**, and **long term**.

1. **Long term schedulers** are responsible for admitting processes into the system and terminating them.
2. **Medium term schedulers** are responsible for suspending and resuming processes.
3. **Short term schedulers** are responsible for allocating CPU time and dispatching processes.

We will be primarily discussing the short term scheduler in this section since it is a core component to implementing a multitasking system. So, our goal here for the demo is to create a **short term scheduler**.

Scheduling Algorithms

There are many different algorithms that we can use, some more complicated then others. While we will provide an introduction to the more common algorithms, we will be sticking with the **Round Robin** approach to keep the demo simple.

First Come First Serve

In **First Come First Serve (FCFS)**, jobs are executed as they come. The algorithm is as simple as its name implies; the scheduler selects the first job and lets it run. Then the second. Then the next, and so on. The algorithm cycles through the jobs in the Ready Queue in the order that they came in. New jobs are not started until the previous one terminates. It is not very well suitable for preemptive multitasking.

Example. In the following example, P1 arrives at time 0, P2 arrives at time 1, and P3 arrives at time 2. these processes are placed in the **Ready queue** to be executed. P1 is the first job, so the algorithm selects it to be run. P2 is selected next, but only after P1 is completed. P2 does not get selected until time=5.

Process	Arrive	Run time	Service time
P1	0	5	0
P2	1	3	5
P3	2	8	8

Shortest Job First

In the **Shortest Job First (SJF)** algorithm, the system must have a way to know the amount of time necessary for each job to execute. The algorithm selects the next job from the **Ready Queue** to execute that has the smallest time delta. This algorithm suffers from the problem of **process starvation**. Jobs can be left in the **Ready Queue** when jobs of smaller time deltas are given priority. Since this example is very similar to the **FCFS algorithm** discussed above and is almost never implemented in practice due to its requirement of calculating time deltas (your software needs to be an oracle to know beforehand how long processes will execute) we do not think another example is needed.

Priority Queue

The system can assign each job a **priority number**. Jobs with higher priority are then selected first. This is the basic idea behind **priority scheduling algorithms**. How priority is determined is up to the designer. Similarly, how to handle the case when two priority are the same is up to the designer. One idea is to have a default priority and make it user adjustable. When two priorities are the same, we can use **FCFS** or **SJF** to decide which one to use. Another idea is to calculate priorities based on a **protocol**. The protocol could be assigned by a system administrator or calculated using some measurement of system resources and memory constraints. It is more often common to see priorities used alongside other scheduling algorithms as we will see later on.

To summarize though, just select the job from the **Ready Queue** that has the highest priority. Like with **SJF**, this algorithm suffers from **process starvation** since processes with higher priorities can starve out processes with lower priorities.

Round Robin

The system gives each process a time slice to run called a **quantum**. The system then **preempts** the currently executing process to allow another process to run. Processes are selected in the order that they appear in the **Ready Queue**. Because all processes are allowed to run, this algorithm does not starve any processes. The system is responsible for **context swapping** in order to save and restore the **execution state** of processes as they are selected to run. We will cover **context swapping** later when we cover **multitasking**.

Example. Given processes P1, P2, P3 and a time quantum of 5, the **Round Robin (RR)** algorithm first selects P1 to run. After the quantum time is up, the system **preempts** P1. P1 is moved to the back of the **Ready Queue**. The system saves the **context** of P1. The algorithm selects P2 and the system performs a **context switch**. P2 can now execute.

| | | | | | |

Process	P1	P2	P3	P1	P2	P3
Quantum=5	0	5	10	15	20	25

Multilevel Queue

Instead of using one **Ready Queue** to decide what to run next, why not use **multiple**? The idea is that we can get the both worlds of privileged levels and another scheduling algorithm by combining them into a **multilevel queue**.

The basic idea is that we would have **multiple queues**. And these queues are for **different priorities**. For example, **if you have 5 priority levels, you would have 5 queues**. The algorithm would first select a job to run based on priority from the highest priority queue. If the queue has multiple jobs in it, it uses another algorithm (like **RR**) to decide what to run. You can also use different scheduling algorithms for the different priority queues. This algorithm has the potential for **starving processes** however for the same reason **priority scheduling** does. So we have a great algorithm here, but what can we do to prevent **starving processes**?

Multilevel Feedback Queue

The **Multilevel Feedback Queue** is a modification of the **multilevel queue** to prevent **process starvation**. The problem with the multilevel queue was that, when a process of some priority L is inserted into queue L, we can starve the process by just submitting new jobs where the priority is greater than L. To prevent this, what we can do is ***change the priority of the process***. So we can move processes from one priority queue into another.

In our example above, the process with priority L would be moved to a higher priority queue after some time passes. This will continue until the process reaches the highest priority queue. Thus the process is never starved out. We can also lower the priority of jobs by moving them into lower priority queues which might be useful when important system tasks need to run. The difficulty of implementing multilevel feedback queues is determining *when* processes should be moved. This is the most common algorithm in use by modern operating systems today.

Example. The following is an example of a **multilevel queue**. Here we have three queues, system processes have the highest priority and applications have the lowest priority. Different scheduling algorithms can be used on each of the different queues to select jobs from them. The scheduler selects the highest priority **non-empty** queue. It then uses another algorithm (such as **FCFS** or **RR**) to select a job from that queue. In **multilevel feedback queues**, the system can move processes between different queues. For example, we can move jobs from L3 then L2 then L1 over time, thereby raising its priority so it can run. Thus no process starvation.

Queue Level	Priority Queue
L1	System Processes
L2	Batch Jobs
L3	Applications

7. Multitasking

We have covered a **lot** of material throughout this chapter. And at long last, we can finally get to the main focus of this chapter, **multitasking**. We will be putting everything together into code.

We first covered **process state management** because the **scheduler** and **multitasking** component need to able to select and move processes between different **states**. For example, the **scheduler** often needs to switch processes from **Ready** to **Running**. If you plan to support more advanced paging techniques (such as **page swapping algorithms**), you will need to be able to switch processes to and from a **Suspended** state. The system needs to be able to differentiate between a **Suspended** process and one that is still in memory awaiting a completion signal. Both processes have a **Process Control Block (PCB)** and uses system resources, however **Suspended** processes aren't using memory. We also needed a way to pause processes. We did this by introducing a **Wait** state. As you can see, state management is a critical component to implementing multitasking. This is why we covered this first.

The next thing we looked at was **Process Creation**. We looked in more detail about how it is used with state management. In **Chapter 24**, we implemented a **CreateProcess** function. Recall, that our function loaded a **Portable Executable (PE)** image into memory, mapped it into the virtual address space, and executed it in user mode. We will be building off of this function in this section to create a new process, and add it to the **Ready queue** to be selected by the **Scheduler**.

Then we looked at an introduction to **concurrent programming**. Topics included the **Critical Section** problem, **Mutual Exclusion**, and **Semaphores**. **Concurrency** happens when multiple processes and threads run ***asynchronously***. Concurrent programming provides techniques that we can use to *synchronize* communication between *asynchronous* processes. Concurrent programming is ***hard*** – there is no *right* way to go about it. If you use concurrency, you can *guarantee* that your code has bugs – most of which may never surface for years or decades. We introduced concurrent programming since the topic of this chapter is multitasking. Since shared resources tie close to multitasking (typically in the form of shared libraries, signals, and message passing), we included a brief introduction to it here.

We then looked at an introduction to **Inter-Process Communication (IPC)**. IPC plays a critical role in all but the simplest of operating systems. And systems that support IPC with multitasking require the concurrent programming techniques discussed in this chapter. You have already been using a form of IPC through the use of **system calls**.

Finally we covered **scheduling algorithms**. The Scheduler is the heartbeat of the operating system. It is responsible for selecting processes for running and is a core algorithm in the multitasking system.

Now, *finally*, we will be putting things together as we dive into the world of multitasking operating systems.

As you recall, there are three types of multitasking:

1. Preemptive
2. Non-Preemptive
3. Cooperative

We will focus on preemptive multitasking.

The Plan

We will be using the **Round Robin (RR)** scheduling algorithm. This algorithm requires us to be able to allocate a **quantum** as a resource to the process being selected. So we'll need a **clock**. The system has many different types of clocks:

1. Programmable Interval Timer (PIT)
2. Advanced Programmable Interrupt Controller (APIC) timer
3. Real Time Clock (RTC)
4. High Performance Event Timer (HPET)
5. etc.

For the purposes of the demo, we will be sticking with the PIT since it has been covered and already supported. So we have our scheduling algorithm and clock that we will be using. In Chapter 24, we introduced the **Process Control Block (PCB)** and **Thread Control Block (TCB)**. We will expand the TCB to include information needed to store the current thread state and switch from user mode to kernel mode.

```
typedef struct _thread {
   uint32_t    esp;
   uint32_t    ss;
   uint32_t    kernelEsp;
   uint32_t    kernelSs;
   struct _process*  parent;
   uint32_t    priority;
   int         state;
   ktime_t     sleepTimeDelta;
}thread;
```

We will need some lower level stuff to create the task associated with a thread. The **stack** stores the current **register context**. We will be storing the register context on the stack pointed to be the **esp** field in the above structure. The **scheduler** is responsible for **creating** tasks, **managing** tasks, and **switching** tasks. We will look at each of these in more detail in the following sections. As always, all sample code is used in the demo program at the end of this chapter.

The Ready Queue

We first need a place to store these tasks. Tasks should be dynamically allocated from a non-paged pool by the kernel memory allocator. However, since the series does not implement a kernel allocator, we are limited to using an array for our implementation. Using a circular queue, we can implement the **First-In-First-Out** functionality required for **Round Robin** scheduling. The idea is so that we can move to the next task by simply removing the top element of the queue and pushing it to the back. So the new task would become the top of the queue.

```
thread   _readyQueue  [THREAD_MAX];
int      _queue_last, _queue_first;
thread   _idleThread;
thread*  _currentTask;
thread   _currentThreadLocal;

/* clear queue. */
void clear_queue() {
        _queue_first = 0;
        _queue_last  = 0;
}

/* insert thread. */
bool queue_insert(thread t) {
        _readyQueue[_queue_last % THREAD_MAX] = t;
        _queue_last++;
        return true;
}

/* remove thread. */
thread queue_remove() {
        thread t;
        t = _readyQueue[_queue_first % THREAD_MAX];
        _queue_first++;
        return t;
}

/* get top of queue. */
thread queue_get() {
        return _readyQueue[_queue_first % THREAD_MAX];
}
```

For our example, we only implement a single queue for ready tasks. Tasks can be removed and added at any time by shuffling the queue around. Notice the **_currentTask** pointer. For Chapter 25, this pointer always points to **_currentThreadLocal** which stores a local copy of the currently executing thread. Our ISR will use the pointer to save and restore the thread state. We will look at the ISR in the next section.

The Interrupt Service Routine (ISR)

Alright, so our first task is to somehow get the scheduler called whenever a timer even triggered. Recall that hardware interrupts are raised by the Interrupt Controller, in our case, the legacy **Programmable Interrupt Controller (PIC)**. There are of course others (such as **Advanced PIC (APIC)** used with **MultiProcessor (MP)** and inter-CPU IRQ's) however we supported the legacy PIC interface only for the series in order to keep things simple. The PIC raises a signal to the CPU when a hardware device sends it to the PIC, such as the IR#0 signal sent from the PIT. The PIC then notifies the CPU by raising another signal, in this case the IRQ line on the CPU. What IRQ that gets called depends on how we programmed the PIC. Recall that we programmed the PIC to map IR#0 to ISR 33. What this means is that, whenever the PIT fires, the CPU stops executing the current code, pushes the return cs, eip, and flags on the current stack, and then calls the ISR that we installed in the **Interrupt Descriptor Table (IDT)**, that is, **IDT[33]**.

In short, we already installed our timer ISR to interrupt vector 33. We did this back when setting up protected mode. It was needed in order for us to enable hardware interrupts. That is fine and all, but what we want to do is ***override*** it.

We do this through ***interrupt chaining***. We introduced interrupt chaining in an earlier chapter, however we never really put it into practice. Until now, that is. What we need to do is to get the old ISR, install our own. Lets do that now,

```
/* register isr */
old_isr = getvect(32);
setvect (32, scheduler_isr, 0x80);
```

Simple enough. We implemented **getvect** and **setvect** back when we talked about the **IDT**. We install it to **IDT[32]** because that is where the **PIT** ISR was. So what this does is save it in **old_isr** and install a new ISR, **scheduler_isr**.

So, with the above in mind, every time the PIT fires, **scheduler_isr** will be called instead. Now for the hard part – writing the ISR. Consider what the ISR needs to do and when it can be called. ***The ISR can be called at any time***. However, ***it is always called when a task is running***. All we need to do is save the current register state and call the scheduler. Do not forget to send the **End-Of-Interrupt (EOI)** to the PIC.

We will first present the ISR implemented for the demo, and then we will break it piece by piece to cover the details of what its doing below.

```
__declspec(naked) void _cdecl scheduler_isr () {
        _asm {
        ;
        ; clear interrupts and save context.
        ;
        cli
        pushad
        ;
        ; if no current task, just return.
        ;
        mov eax, [_currentTask]
        cmp eax, 0
        jz  interrupt_return
        ;
        ; save selectors.
        ;
        push ds
        push es
        push fs
        push gs
        ;
        ; switch to kernel segments.
        ;
        mov ax, 0x10
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        ;
        ; save esp.
        ;
        mov eax, [_currentTask]
        mov [eax], esp
        ;
        ; call scheduler.
        ;
        call scheduler_tick
        ;
        ; restore esp.
        ;
        mov eax, [_currentTask]
        mov esp, [eax]
        ;
        ; Call tss_set_stack (kernelSS, kernelESP).
        ; This code will be needed later for user tasks.
        ;
        push dword ptr [eax+8]
        push dword ptr [eax+12]
        call tss_set_stack
        add esp, 8
        ;
        ; send EOI and restore context.
        ;
        pop gs
        pop fs
        pop es
        pop ds
interrupt_return:
        ;
        ; test if we need to call old ISR.
        ;
        mov eax, old_isr
        cmp eax, 0
        jne chain_interrupt
        ;
        ; if old_isr is null, send EOI and return.
```

```
        ;
        mov al,0x20
        out 0x20,al
        popad
        iretd
        ;
        ; if old_isr is valid, jump to it. This calls
        ; our PIT timer interrupt handler.
        ;
chain_interrupt:
        popad
        jmp old_isr
        }
}
```

The ISR is responsible for **saving the current register context** and **saving the stack pointer of the current task**. It then **calls the scheduler**, and **restores the stack pointer from the current task** and **restores the register context that we saved before**. Since everything is restored, the task and continue executing without problems when the ISR returns. The ISR appears more complicated then it actually is. Let's take a closer look at it in pieces. Like all of our other ISR's, the very first thing we do is save the current register state in order to preserve them on the stack. So the ISR begins like this:

```
__declspec(naked) void _cdecl scheduler_isr () {
        _asm {
        cli
        pushad

        popad
        iretd
        }
}
```

Since we install the ISR on top of the ISR that was installed by the PIT, we need to be very careful here. This means that **our scheduler_isr will be called with every clock tick**. When we call **setvect** to install it, the **PIT can fire before we have any tasks in the ready queue**. When there are no tasks to run, we just want the ISR to return since there is nothing to do. You might also notice that we disable interrupts but never restore them. This is fine. Currently running tasks enable interrupts through the FLAGS register. Since the FLAGS register is preserved in all cases, when we issue IRETD, FLAGS.IF will enable when we return thereby re-enabling interrupts. Our ISR becomes,

```
__declspec(naked) void _cdecl scheduler_isr () {
        _asm {
        cli
        pushad
        ;
        ; if no current task, just return.
        ;
        mov eax, [_currentTask]
        cmp eax, 0
        jz  interrupt_return

        ;
        ; <actual ISR code here>
        ;

interrupt_return:
        popad
        iretd
        }
}
```

Finally, we need to keep in mind that the PIT hardware is now calling **scheduler_isr**, so the PIT driver ISR is never being called. We want to **chain the interrupt**. This means, if there is an old ISR that was installed before us, we want to give it a chance to run. This is done by **jumping** (not calling) to it. When calling another ISR, we need to keep in mind that the ISR will either chain another interrupt or issue an **End-Of-Interrupt (EOI)** command to **break the chain**. When calling another ISR, we are still technically servicing an interrupt, so don't want to send EOI nor do we need an IRETD. However, when not calling another ISR and giving control back to the original process, we need both. So our ISR now becomes:

```
__declspec(naked) void _cdecl scheduler_isr () {
        _asm {
        ;
        ; clear interrupts and save context.
        ;
        cli
        pushad
        ;
        ; if no current task, just return.
        ;
        mov eax, [_currentTask]
        cmp eax, 0
        jz  interrupt_return

        ;
        ; <actual ISR code here>
        ;

interrupt_return:
        ;
        ; test if we need to call old ISR.
        ;
        mov eax, old_isr
        cmp eax, 0
        jne chain_interrupt
        ;
        ; if old_isr is null, send EOI and return.
        ;
        mov al,0x20
        out 0x20,al
        popad
        iretd
        ;
        ; if old_isr is valid, jump to it. This calls
```

```
        ; our PIT timer interrupt handler.
        ;
chain_interrupt:
        popad
        jmp old_isr
        }
}
```

The actual body of the ISR that performs the actual tasking is the following part:

```
        ;
        ; save selectors.
        ;
        push ds
        push es
        push fs
        push gs
        ;
        ; switch to kernel segments.
        ;
        mov ax, 0x10
        mov ds, ax
        mov es, ax
        mov fs, ax
        mov gs, ax
        ;
        ; save esp.
        ;
        mov eax, [_currentTask]
        mov [eax], esp
        ;
        ; call scheduler.
        ;
        call scheduler_tick
        ;
        ; restore esp.
        ;
        mov eax, [_currentTask]
        mov esp, [eax]
        ;
        ; Call tss_set_stack (kernelSS, kernelESP).
        ; This code will be needed later for user tasks.
        ;
        push dword ptr [eax+8]
        push dword ptr [eax+12]
        call tss_set_stack
        add esp, 8
        ;
        ; srestore context.
        ;
        pop gs
        pop fs
        pop es
        pop ds
```

It first pushes segment registers on the stack. (Recall that we did a PUSHAD before this. And the CPU pushed CS, EIP, and EFLAGS on the stack as well when the ISR was first called.) We store these on the stack so that we can **save the current thread register context**. **The order that these registers are pushed on the stack matches the order that we use later in the stackFrame structure**. We then set those segment registers to the kernel mode selectors we set up a long time ago from the **Global Descriptor Table (GDT)**. We do this because we are not making the assumption that the currently running task is a kernel mode task. **If the task is a user mode task, DS, ES, FS, and GS would still be 0x23 rather then 0x10**. We saved the original task selectors on the threads stack, so we can adjust them now. The CPU automatically sets SS and CS for us from the Task State Segment (TSS) when coming from a user mode task, so those would already be set appropriately. We will take a little more closer look at the stacks a little later. Finally, **we save the current value of ESP to _currentTask->esp** and call **scheduler_tick**.

_currentTask is assumed by the ISR to always be pointing to whatever the currently running task is. If the scheduler changes tasks, then that new task becomes the new "currently" running task. Even if its a new task, we just restore ESP to that new tasks **_currentTask->esp** field. Since we initially saved the register context on the new threads stack, we just pop them off back into their respective registers. We also call **tss_set_stack** that we implemented a long time ago. This is only useful if the task that we are returning to is a user mode task. What we do is set the new tasks kernel stack into TSS by updating it. For the upcoming demo, we will only be running kernel threads, each with only one kernel stack so this does not apply just yet. However, keep in mind that user level threads have two stacks rather then one, since the threads run in both user space and kernel space. We will be expanding on this farther in the next couple of chapters as we dive into address space management and user space.

So how do we switch tasks? Consider for a moment what would happen if that **_currenTask** pointer changes when the scheduler is called. Since the register context and stack pointer of this new task was saved the same way, by simply changing this pointer inside of the **scheduler_tick** function, the ISR would automatically load the new tasks register context and stack. And so, **task switching is as simple as updating that pointer**.

Switching Tasks

So switching tasks just involves updating a pointer. With Round Robin scheduling, we can use a queue to store the running tasks. Since queues already operate in **First-In-First-Out** order, all we need to do is remove and reinsert the current task to push it back. This greatly simplifies the code.

```
/* schedule next task. */
void dispatch () {

        /* We do Round Robin here, just remove and insert.
        Note _currentTask pointer always points to
        _currentThreadLocal. So just update _currentThreadLocal. */
        queue_remove();
        queue_insert(_currentThreadLocal);
        _currentThreadLocal = queue_get();
}

/* gets called for each clock tick. */
```

```
void scheduler_tick () {

        /* just run dispatcher. */
        dispatch();
}}
```

That is all there is to it. The above implements **Round Robin** scheduling and swaps between the tasks after a certain **quantum** is up. Tasks are stored in the **Ready Queue** which was implemented earlier. This only leaves one more thing – task creation.

Although the above works for multiple threads, it will not work for threads belonging to different processes. The typical solution is to compare the current threads parent process with the new one. If they belong to the same process, then the dispatcher can simply return. If they belong to different processes, the dispatcher needs to invoke the VMM to switch to the new process address space. To keep the example code simple, we opted to avoid this for this chapter. However, we will be supporting it in the next chapter or two when we cover address space management in greater detail.

Task Creation

Let's say that our **schedule** function updates the **_currentTask** pointer to a different task. So when this function returns back to the ISR, the ISR will set the stack and register context from this new task before issuing IRETD. This works well, but only if the task already has a stack and register context on the stack.

So we need to set it up when creating the task in the first time. So we set up a basic stack frame and set the task **esp** and **eip** to the stack and **entry** point function. The stack frame must be one that is expected by our ISR. When we return back to the ISR, it will POP GS, POP FS, POP ES, POP DS first, then does a PUSHA followed by an IRETD. PUSHA pops EAX, EBX, ECX, EDX, ESI, EDI, ESP, and EBP. And IRETD pops EIP, CS, and FLAGS. So this must be our initial stack frame when the task is created.

```
typedef struct _stackFrame {
  uint32_t gs;
  uint32_t fs;
  uint32_t es;
  uint32_t ds;
  uint32_t eax;
  uint32_t ebx;
  uint32_t ecx;
  uint32_t edx;
  uint32_t esi;
  uint32_t edi;
  uint32_t esp;
  uint32_t ebp;
  uint32_t eip;
  uint32_t cs;
  uint32_t flags;
}stackFrame;

task  task_create (uint32_t entry, uint32_t esp) {
  thread t;
  stackFrame* frame = ((stackFrame*) esp);
  frame->flags = 0x202;
  frame->cs    = 8;
  frame->eip   = (uint32_t)entry;
  frame->ebp   = 0;
  frame->esp   = 0;
  frame->edi   = 0;
  frame->esi   = 0;
  frame->edx   = 0;
  frame->ecx   = 0;
  frame->ebx   = 0;
  frame->eax   = 0;
  frame->ds    = 0x10;
  frame->es    = 0x10;
  frame->fs    = 0x10;
  frame->gs    = 0x10;
  t.esp = (uint32_t) frame;
  t.ss = 0x10;
  return t;
}
```

This works for most tasks, except one – the ***initial task***. The ISR we created will only work if the currently executing code is within a task. It is yet another chicken and egg problem. To get around this, we need to create a special task object and execute it when we are ready to start multitasking.

```
static thread _idleTask;
void task_execute(thread t) {
  _asm{
    mov esp, t.esp
    pop gs
    pop fs
    pop es
    pop ds
    popad
    iretd
  }
}

/* initialize scheduler. */
void scheduler_initialize(void) {

        /* clear ready queue. */
        clear_queue();

        /* clear process list. */
        init_process_list();

        /* create idle thread and add it. */
        _idleThread = thread_create(idle_task, (uint32_t) create_kernel_stack(), true);

        /* set current thread to idle task and add it. */
        _currentThreadLocal = _idleThread;
        _currentTask        = &_currentThreadLocal;
```

```
        queue_insert(_idleThread);

        /* register isr */
        old_isr = getvect(32);
        setvect (32, scheduler_isr, 0x80);
}

/* idle task. */
void idle_task() {
  while(1) _asm pause;
}
```

The above puts everything together. It creates an idle task, adds it to the queue, installs the ISR, and executes the initial task. When the initial task executes, the ISR will be called whenever the PIT fires to call the scheduler to update the current task if needed.

8. Introduction to MP

This is a *very* brief introduction to the **Multi-Processor (MP) Specification** that is designed to provide a standard interface for starting up the other processors and **Inter-Processor Interrupts (IPI)**. We consider this an advanced topic since it can quickly escalate the difficulty of concurrent programming. Our scheduler only executes one task at a time, however with MP, we can implement a low level scheduler responsible for scheduling independent CPU's for the tasks and can achieve running multiple tasks at the same time. So what we are presenting here is just a very brief introduction – for anyone wanting to dive more into the MP standard, we recommend checking out the MP specification. Your system must already support the IOAPIC, LAPIC, and ICI which are used by MP.

There is **symmetric multiprocessing (SMP)** and **asymmetric multiprocessing (ASP)**. In SMP, all of the processors are of the same type whereas in ASP they are not. Most systems only support SMP given that ASP systems are very rare in desktop systems. However, the MP standard is applicable to both and gives itself some room for extendability so it can adopt to more diverse machine types and farther allows the operating systems to adopt and configure itself for different types of systems.

When the system first starts up, the hardware selects a **Boot-Strap Processor (BSP)** to act as the sole processor to start up. The BSP is the first processor to start up and must be the last processor to shut down. The operating system may send a **STARTUP IPI** from the BSP to another **Application Processor (AP)** to start it. Other AP's can be started by either the BSP or another AP. The **STARTUP IPI** (and **INIT IPI**) is what the operating system sends to wake the other processors.

The operating system must first search for the floating **MP Floating Pointer** structure in order to detect if the system supports MP. The structure contains the **physical address** of the **MP Configuration Table**. The configuration table is **read only**. It stores the **memory mapped address of the Local APIC (LAPC), Processor entries (including the processor LAPIC ID), IOAPIC entries (including IOAPIC base memory mapped address), Buses, and interrupt configuration entries**. The operating system must remember the LAPIC ID of the BSP to make sure it is the last one to shut down.

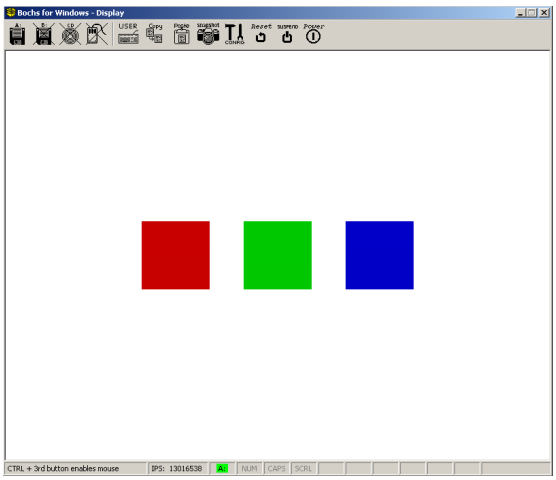
To wake another AP, all we need to do is send a **INIT IPI** through the BSP LAPIC or another AP LAPIC. The memory mapped registers for the LAPIC's are stored in the processor information in the MP configuration table. We then need to send an **STARTUP IPI** to that AP to start executing. That's really all there is to it. The INIT IPI causes the AP to **reset**. The STARTUP IPI causes it to start executing at the location you tell it to in real mode. Operating systems must provide a real mode stub routine for configuring the API's in protected or long modes just as you did for the BSP.

So that is pretty much all we wanted to cover in this brief introduction to multiprocessor systems. Starting up other processors (or processor cores0 is fairly simple and we encourage experimenting with SMP after implementing your scheduler. We may cover MP in more detail in a later tutorial after covering the APIC. We just wanted to give a little overview and direction for those interested in it now.

9. Demo

[Download Demo]

Most of the new code has been covered in the above text, we are just preparing the initial release. Assuming no problems arise during stress tests and final integration, the demo should be released sometime within the next week or two.



Demo running in 800x600x32 mode executing three tasks.

This is our first real graphical demo. The demo runs three tasks concurrently; each task cycles through a select color in video memory while running to visually show that they are executing. We opted to have a graphical demo rather then text based since we

believe we can have it more visually appealing yet still simple to do. It is alright if you did not read through the graphics series yet, we will discuss things here.

Bochs Graphics Adapter (BGA)

To keep the code as simple as possible so we can focus on the primary topic of the chapter, we opted to use BGA under the assumption that the system is configured for ISA. ***This code is Bochs specific*** and will not work on real systems. Real systems would require scanning the PCI bus infrastructure which may be a topic in a more advanced chapter.

```
#define VBE_DISPI_IOPORT_INDEX          0x01CE
#define VBE_DISPI_IOPORT_DATA           0x01CF
#define VBE_DISPI_INDEX_XRES            0x1
#define VBE_DISPI_INDEX_YRES            0x2
#define VBE_DISPI_INDEX_BPP             0x3
#define VBE_DISPI_INDEX_ENABLE          0x4
#define VBE_DISPI_DISABLED              0x00
#define VBE_DISPI_ENABLED               0x01
#define VBE_DISPI_LFB_ENABLED           0x40

void VbeBochsWrite(uint16_t index, uint16_t value) {
   outportw (VBE_DISPI_IOPORT_INDEX, index);
   outportw (VBE_DISPI_IOPORT_DATA, value);
}

void VbeBochsSetMode (uint16_t xres, uint16_t yres, uint16_t bpp) {
   VbeBochsWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_DISABLED);
   VbeBochsWrite (VBE_DISPI_INDEX_XRES, xres);
   VbeBochsWrite (VBE_DISPI_INDEX_YRES, yres);
   VbeBochsWrite (VBE_DISPI_INDEX_BPP, bpp);
   VbeBochsWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_ENABLED | VBE_DISPI_LFB_ENABLED);
}
```

To set the video mode just involves calling **VbeBochsSetMode**. We use **800x600x32** in our example since it appears to be well supported. **The Linear Frame Buffer (LFB)** under ISA is at the predefined location **0xe0000000**. However, because we have paging enabled, we need to map the LFB into our virtual address space to use it. We will map it to 0x200000 virtual for the demo. The mapping is done by calculating the size of the LFB in number of pages, and mapping each page by calling our VMM.

```
void* VbeBochsMapLFB () {

/* BGA LFB is at LFB_PHYSICAL for ISA systems. */
#define LFB_PHYSICAL 0xE0000000
#define LFB_VIRTUAL  0x200000

  /* map LFB into current process address space. */
  int pfcount = WIDTH*HEIGHT*BYTES_PER_PIXEL/4096;
  int c;
  for (c = 0;c <= pfcount; c++)
    vmmngr_mapPhysicalAddress (vmmngr_get_directory(),LFB_VIRTUAL + c * 0x1000,LFB_PHYSICAL + c * 0x1000, 3);

  /* return pointer to LFB. */
  return (void*) LFB_VIRTUAL;
}
```

With the above function, we can now draw to the LFB by writing to 0x200000. To clean up any possible garbage on the display, we clear it next. Since we need to draw a lot of pixels, we try to optimize the function for 32 bit modes. This function makes the screen white.

```
void fillScreen32 () {
  uint32_t* lfb = (uint32_t*) LFB_VIRTUAL;
  for (uint32_t c=0; c<WIDTH*HEIGHT; c++)
    lfb[c] = 0xffffffff;
}
```

In 32 Bits Per Pixel modes, the pixel colors are composed of 8 bits for red, 8 bits for green, and 8 bits for blue. The high 8 bits are ignored for our purposes but is typically used as a transparency value. We use three separate tasks to render the three rectangles and cycle through the intensity of the three colors. Since we are going to render to different locations on display, we won't have to worry about concurrency problems here. Although display memory is shared, each task will render to separate parts.

```
void rect32 (int x, int y, int w, int h, int col) {
  uint32_t* lfb = (uint32_t*) LFB_VIRTUAL;
  for (uint32_t k = 0; k < h; k++)
    for (uint32_t j = 0; j < w; j++)
      lfb[(j+x) + (k+y) * WIDTH] = col;
}
```

```
void kthread_1() {
  int col = 0;
  bool dir = true;
  while(1) {
    rect32(200,250,100,100,col << 16);
    if (dir){
      if (col++ == 0xfe)
        dir=false;
    }else
      if (col-- == 1)
        dir=true;
  }
}
```

```
void kthread_2 () {
  int col = 0;
  bool dir = true;
  while(1) {
    rect32(350,250,100,100,col << 8);
    if (dir){
      if (col++ == 0xfe)
        dir=false;
    }else
      if (col-- == 1)
        dir=true;
  }
}
```

```
void kthread_3 () {
  int col = 0;
  bool dir = true;
  while(1) {
    rect32(500,250,100,100,col);
    if (dir) {
      if (col++ == 0xfe)
        dir=false;
    }else
      if (col-- == 1)
        dir=true;
  }
}
```

Thread Stacks

Typically a thread has two separate stacks. One for when executing in user mode, and another for when executing in kernel mode. Recall that when a thread is executing in user mode, the CPU switches to a kernel stack by getting the **esp0** and **ss0** fields of the **Task State Segment (TSS)**. The scheduler is responsible for updating the TSS to the new threads kernel mode stack. However, for chapter 25, since all threads run in kernel space, the TSS will never be referenced. In other words, **the threads in chapter 25 only have one stack – a kernel mode stack**.

We will be supporting user mode threads within the next two chapters when we cover address space management. We will use our future address space allocator to reserve stack space in user space for each user mode thread. That means **threads will have both a user mode and kernel mode stack**.

The thread uses the kernel mode stack when executing code with **Current Privilege Level (CPL)** of 0. The CPU automatically loads this if the CPL is less then the **Requested Privilege Level (RPL)** from the **TSS**. In other words, lets say that our user mode thread is running and the PIT fires. The CPU will then set **SS=TSS.ss0** and **ESP=TSS.esp0**. It will then **push the return CS and IP on this new stack** and call the ISR. When the ISR is done, it executes **IRET** to return back to the user mode code and stack.

This is why user level threads must have, at a minimum, ***two*** separate stacks. The first stack must be mapped in kernel space and the other must be mapped in user space so the program can access it while executing. Kernel level threads only need ***one*** stack.

Since we don't have an address space allocator, we cannot nicely allocate user mode stacks just yet, so cannot support user level threads (without hacks.) And since we don't have a proper kernel mode allocator yet, we can't nicely support allocation of kernel level stacks either. These will be the topics for the next chapter or two.

So what we decided to do for chapter 25 was to reserve space in kernel memory and allocate each 4k block as its own stack.

```
void* create_kernel_stack() {

        physical_addr       p;
        virtual_addr        location;
        void*               ret;

        /* we are reserving this area for 4k kernel stacks. */
#define KERNEL_STACK_ALLOC_BASE 0xe0000000

        /* allocate a 4k frame for the stack. */
        p = (physical_addr) pmmngr_alloc_block();
        if (!p) return 0;

        /* next free 4k memory block. */
        location = KERNEL_STACK_ALLOC_BASE + _kernel_stack_index * PAGE_SIZE;

        /* map it into kernel space. */
        vmmngr_mapPhysicalAddress (vmmngr_get_directory(), location, p, 3);

        /* we are returning top of stack. */
        ret = (void*) (location + PAGE_SIZE);

        /* prepare to allocate next 4k if we get called again. */
        _kernel_stack_index++;

        /* and return top of stack. */
        return ret;
}
```

Back to Sleep()

You might recall that we implemented a very basic **sleep** function that we needed in order to delay the read operation of the floppy device. Our implementation simply went into a **busy loop** in order to waste some time. Now we can adopt it for the threading system.

The basic idea is that **sleep** should **pause** the thread that called the function. This means we need to adjust the current thread state from READY to BLOCK and force a task switch. The scheduler then needs to keep track of blocked threads to handle them properly. This is typically done via **Signals** from other operating system components. For example, if a thread is waiting for a device to be ready, it may block. Now the system needs to wait until that thread receives a signal from the driver. Until then, the scheduler should jump to executing other threads. To keep the demo relatively simple, we opted to do things a little differently.

All we need to do is change the state of the currently running program and force a task switch (by calling the ISR directly via **int 33**.) The scheduler would contain the logic code for checking blocked threads while selecting new threads to run. If the next thread is blocked, we decrement its sleep time delta and awake the thread is the sleep time delta reaches zero.

Although we do not use sleep in this demo, the disk driver code relies on it. So now the thread attempting to read from the disk device can properly sleep.

Main Program

Finally, we will take a look at the main program. In the demo for Chapter 25, we moved the stack into kernel space and readjust it after making a static copy of the boot parameter block that was passed from the boot loader. We then use the services discussed above to set the video mode, initialize the scheduler, and create and add three threads to the ready queue. Since the threads run in kernel space, they only have a kernel stack allocated to them, which we allocate calling **create_kernel_stack**.

We have also completely rewritten the process creation and management code from Chapter 24 to be compatible with the thread system created in Chapter 25. However, it will not be completed until we support the allocation of user mode stacks which we will do in the upcoming chapters.

```
void _cdecl kmain (multiboot_info* bootinfo) {

        /* store kernel size and copy bootinfo. */
        _asm mov        word ptr [kernelSize], dx
        memcpy(&_bootinfo, bootinfo, sizeof(multiboot_info));

        /* adjust stack. */
        _asm lea esp, dword ptr [_kernel_stack+8096]
        init (&_bootinfo);

        /* set video mode and map framebuffer. */
        VbeBochsSetMode(WIDTH,HEIGHT,BPP);
        VbeBochsMapLFB();
        fillScreen32 ();
```

```
        /* init scheduler. */
        scheduler_initialize ();

        /* create kernel threads. */
        queue_insert (thread_create(kthread_1, (uint32_t) create_kernel_stack(),true));
        queue_insert (thread_create(kthread_2, (uint32_t) create_kernel_stack(),true));
        queue_insert (thread_create(kthread_3, (uint32_t) create_kernel_stack(),true));

        /* execute idle thread. */
        execute_idle();

        /* this should never get executed. */
        for (;;) _asm {cli
                hlt};
}
```

10. Conclusion

In this chapter we looked at scheduling algorithms, a brief overview of SMP, concurrent programming, and implemented a working preemptive Round Robin scheduler. We have also went through a small introduction to high resolution video modes using Bochs Graphics Adapter (BDA), state management, and an introduction to several IPC techniques.

In the next chapter(s), we will finally cover memory allocation algorithms, including kernel and user mode allocators, address space allocations, page swapping and page fault handling. Topics will include the free list and stack allocators, SLAB allocator (and possibly its variants), Zone and Arena allocators, Buddy allocators, user space management, recursive page directories, page files and swap space, and possibly others. We will expand on the material from this chapter to support user mode process loading. Due to the amount of material coming up, this may be one or two separate chapters.

Until next time,

~Mike ();

OS Development Series Editor

Home

Operating Systems Development Series

Operating Systems Development - Portable Executable (PE)

by Mike, 2011

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

Welcome!

Yey, this is going to be a long one.

This chapter is going to cover an advanced topic - the PE executable file format. We will be looking at covering PE resources, dynamic linking, and more. This chapter is also planned for an update to include more information to make the information as complete as possible.

Most of what is included in this chapter is for information purposes only and are only included both for completness and in case any of our readers would like to provide support for them. Also please note that a lot of the information provided can also be found in the official PE specification.

After this chapter, we will have everything we need to develop a loader and support a single tasking environment.

Lets get started!

File Format

Abstract

The **Portable Executable (PE)** file format is the standard executable file format used in several operating systems, including Windows and Windows-like OSs, such as ReactOS. It is also the standard file format used with booting on **Extensable Firmware Interface (EFI)** machines.

The PE executable file format is a complex format, supporting relocations, symbol tables, resources, dynamic binding, and more.

Terms

VA (Virtual Address)

A **Virtual Address (VA)** is a linear address in the **Virtual Address Space (VAS)** of the current program. All addresses in the PE executable format are virtual addresses. These addresses are 32 bit linear addresses.

RVA (Relative Virtual Address)

A **Relative Virtual Address (RVA)** is a VA that is relative to the **base address** of the executable program. The PE executable format uses RVAs in a lot of areas, so it is important to know what RVAs are and how to obtain linear addresses from them. RVAs are just offsets from the base address, thats all. So to obtain its linear address, just add the RVA to the base:

```
Linear address = Base address + RVA
```

This one is important as we will need to perform this calculation in a lot of areas when parsing.

Sections and the Section Table

Sections

Advanced executable file formats typically use **program sections** to simplify the linking process and provide structure to the software. Sections simplifies the linking process by providing a standard method for instructions and data to be stored within the executable image or object file.

A section typically has a name associated with what elements are inside of the section. For example, **.data** is a common section name that contains variable, uninitialized data. Other section names have historical backgrounds. For example, **.text** is a typical name of a section containing executable or object code. **.bss** is typically used for global, program-wide initialized data.

Using the C++ toolchain, for an example, variables defined in the global namespace or as **static** are stored in **.bss**. The resulting bytecode generated after compilation is stored in **.text**.

The PE executable file format typically includes one or more of the following section names:

- .text
- .data
- .bss
- .arch
- .edata
- .idata
- .pdata
- .rdata
- .reloc
- .rsrc
- .sbss
- .sdata
- .srdata
- .xdata

Section Table

Program files and object files contain multiple sections. The base location of each section, and the name of that section, is typically stored in a **section table**. In some implimentations, a section table can be a simple linked list of structures or a hash table - different implimentations exist.

Symbols and the Symbol Table

Symbols

While programming in C++, you most likley have encountered the unfamous **Undefined symbol** linker errors (or in some implimentations of C, warnings. Thats right, fully compiling and linking without error *ahem* old MSVC). This happens do to calling a function or referencing a variable by name whose definition could not be resolved during the linking stage.

Functions and variables are referred to as **symbols** by the linker. A **symbol** contains the name, and information about what it is: such as a data type and value, for example. During compiling, the compilier must keep track of these symbols to insure that the final program can be linked. If a symbol is used that is not defined in the current **translation unit**, but is an EXTERN symbol, the compilier will need to mark it as an EXTERN symbol when writing the symbols to the object file.

If, during the linking stage, a symbol marked EXTERN still has no value associated with it (the symbol has no definition), the linker issues the above error.

Symbols are what allows programmers to define variables or functions across modules, translation units, or libraries.

There can only be one symbol with the same name through the entire program, and any libraries that it links with. Because of this, and the high probability of naming collisions with the use of high level languages, variable and function names typically have **name mangling** applied. This, of course, does not apply to assembly language. The name mangling applied depends on multiple factors, and differ between toolchains.

Lets take a look. Here are some C function declarations, and their mangled symbolic name on the right. The number in the mangled name is the number of bytes for the paramaters.

```
void _cdecl function (int i);   ->   _function
void _stdcall function(int i);  ->   _function@4
void _fastcall function(int i); ->   @function@4
```

Notice that functions with the _cdecl call convention only have an underscore prepended. This allows C functions to be easily defined using assembly language, and allows C code to easily call those functions.

There isnt any standard for C++ name mangling. Some compiliers might produce a symbolic name like **?h@@YAXH@Z**, while others can produce **__7h__Fi** or **W?h$n(i)v** for the **same** function of **void h(int)**. This makes it impractical to use with assembly language. It is still possible, however.

Symbol table

Simular to the section table there exists a **symbol table**. The symbol table allows a way for the software to look up symbolic names and information about the symbol, such as if its an exported symbol, its data type, properties, etc. Symbol tables are typically a linked list of information or implimented in hash tables.

Structure

Abstract

We have taken a look at the structure of the PE executable format in our MSVC++ chapter. When we load a PE executable in memory, that memory would contain an exact copy of our loaded file. This means the first byte within the first structure of the PE file format is actually located at the first byte from where the file was loaded in memory.

For example, if we load a PE file to 1MB, the in-memory footprint might look like this:

Notice that the file contents directly start at where it was loaded in memory. This means, in order to read file contents, read it from the loaded image file.

The above image should look famailier to our readers who have read the MSVC chapter. Looking at the above image, if the PE file was loaded to 1MB, then the first on-disk structure, IMAGE_DOS_HEADER, begins at that location in memory, followed by the rest of the structures in the file (including padding).

The above image is also an oversimplification - it does not, by any means, show the complete PE file format. The structure of the PE file format is fairly large, composed of a lot of structures and tables.

Here is the complete format:

1. IMAGE_DOS_HEADER structure **(Important)**
2. STUB program
3. IMAGE_FILE_HEADER structure [COFF Header] **(Important)**
4. IMAGE_OPTIONAL_HEADER structure **(Important)**
5. Segment Table
6. Resource Table
7. Resident Name Table
8. Module Reference Table
9. Imported Names Table
10. Entry Table
11. Non Resident Name Table
12. Segments
    1. Data
    2. Info

The above table lists the complete file format - from beginning to end. Items marked **important** are required to know how to parse in order to just execute the program. All of the other information if provided for information purposes only. The only important members in **IMAGE_OPTIONAL_HEADER** structure are the member that contains the address of the entry point, and image base address.

We will cover parsing each section of this file in detail in the upcoming sections. We will also introduce the other structures used when parsing tables and directories as well.

IMAGE_DOS_HEADER structure

IMAGE_DOS_HEADER is the first structure of the PE file. It contains global information about the program file and how to load it. Most of the information contained in this structure were more relivant to DOS software, and are only supported for backward compatability.

The structure follows the format:

```
typedef struct _IMAGE_DOS_HEADER {  // DOS .EXE header
    uint16_t e_magic;           // must contain "MZ"
    uint16_t e_cblp;            // number of bytes on the last page of the file
    uint16_t e_cp;              // number of pages in file
    uint16_t e_crlc;            // relocations
    uint16_t e_cparhdr;         // size of the header in paragraphs
    uint16_t e_minalloc;        // minimum and maximum paragraphs to allocate
    uint16_t e_maxalloc;
    uint16_t e_ss;              // initial SS:SP to set by Loader
    uint16_t e_sp;
    uint16_t e_csum;            // checksum
    uint16_t e_ip;              // initial CS:IP
    uint16_t e_cs;
    uint16_t e_lfarlc;          // address of relocation table
    uint16_t e_ovno;            // overlay number
    uint16_t e_res[4];          // resevered
    uint16_t e_oemid;           // OEM id
    uint16_t e_oeminfo;         // OEM info
    uint16_t e_res2[10];        // reserved
    uint32_t   e_lfanew;        // address of new EXE header
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;
```

Alright, a lot of interesting things in this structure. The initial CS:IP and initial SS:SP members should be ignored as the operating system normally allocates a stack space and code descriptor value for CS. These members were more prominent during the DOS area and software requiring v8086 mode.

STUB Program

Okay then! Lets look back up at the PE file image structure again (The above picture.) Notice how a DOS stub program is right after the **IMAGE_DOS_HEADER** structure. This is a useful program, actually. This is the program that displays "This program cannot be run in DOS Mode", if you try to execute a Windows program from within DOS.

We can change the stub program by using the **/STUB** linker option:

```
/stub=myprog.exe
```

When DOS attempts to load the executable, it will parse the IMAGE_DOS_HEADER structure and attempt to execute the DOS stub program because it is a valid DOS program. When running under the Win32 subsystem, the Windows loader will ignore the stub program.

IMAGE_NT_HEADERS

Following the STUB program is a structure, **IMAGE_NT_HEADERS** that contains the format of the PE header structures. Here is the structure:

```
typedef struct _IMAGE_NT_HEADERS {
  DWORD                 Signature;
  IMAGE_FILE_HEADER     FileHeader;
  IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS, *PIMAGE_NT_HEADERS;
```

The signiture must match with "PE\0\0", where \0\0 are null characaters. The **IMAGE_FILE_HEADER** contains additional information used by the loader and the complete size of the **IMAGE_OPTIONAL_HEADER** structure. The **IMAGE_OPTIONAL_HEADER** is the largest and most important structure in the file. It also does not have a defined size.

In order to locate this structure, the OS loader must use the **e_lfanew** member of IMAGE_DOS_HEADER. **e_lfanew** is an RVA to this structure in memory, so in order to locate this structure, the loader needs to perform the following:

```
IMAGE_DOS_HEADER* pFile = (IMAGE_DOS_HEADER*) imageBase;
IMAGE_NT_HEADERS* pHeaders = (IMAGE_NT_HEADERS*) (pFile->e_lfanew + imageBase);
```

This assumes **imageBase** referres to the location where the program file was loaded into memory. Because older operating systems, such as DOS are not aware of this member of the header, it will be ignored by these OSs.

This structure contains the format for the other two header structures. Lets look at the first of these structures.

IMAGE_FILE_HEADER

The **IMAGE_FILE_HEADER** is the **Common Object File Format (COFF)** header structure. It follows the following format:

```
typedef struct _IMAGE_FILE_HEADER {
    USHORT  Machine;
    USHORT  NumberOfSections;                   // Number of sections in section table
    ULONG   TimeDateStamp;                      // Date and time of program link
    ULONG   PointerToSymbolTable;               // RVA of symbol table
    ULONG   NumberOfSymbols;                    // Number of symbols in table
    USHORT  SizeOfOptionalHeader;               // Size of IMAGE_OPTIONAL_HEADER in bytes
    USHORT  Characteristics;
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

This structure isnt too complex. Most of the above is only useful for debuggers (symbol table parsing). **SizeOfOptionalHeader** is important - because IMAGE_OPTIONAL_HEADER does not have a defined size, this member lets you know the size of the structure.

Machine can be one of the following values:

- 0x014c for x86 machines
- 0x0200 for x64 machines
- 0x8664 for AMD64 machines

In the usual case, it should be **0x014c** as we are developing for the x86 architecture.

Characteristics is composed of bit flags that can be bitwise-ORd by the linker to let the loader know different properties of the type of executable image this is. Heres the format:

- **Bit 0**: If set, image has no relocation information
- **Bit 1**: If set, File is executable
- **Bit 2**: If set, image has no COFF line numbers
- **Bit 3**: If set, image has no COFF symbol table entries
- **Bit 4**: If set, trim the working set for image. (Windows memory management specific. Obsolete)
- **Bit 5**: If set, loader assumes executable can handle >2GB VAs
- **Bit 6**: If set, loader assumes image supports 32 bit words
- **Bit 7**: If set, image has no debug information
- **Bit 8**: If set, image wont run directly from network drive (Windows specific)
- **Bit 9**: If set, image is treated as a SYSTEM file
- **Bit 10**: If set, image is treated as a DLL file
- **Bit 11**: If set, image will only run on single-processor machines
- **Bit 12**: If set, big-endian. obsolete flag

The Windows headers use defined constants, such as IMAGE_FILE_RELOCS_STRIPPED and IMAGE_FILE_EXECUTABLE_IMAGE that can be used when setting these flags.

As you can see, most of this structure is for information only for the loader on how to load the image. But wait! What about resources, symbol tables, debug info ... where is this at? Ah, behold the reason why **IMAGE_OPTIONAL_HEADER** does not have a defined size. Lets take a look!

IMAGE_OPTIONAL_HEADER

Ugh, here we go. This is the most complex structure in the file. The good news is that you probably have seen this structure before:

```
struct _IMAGE_OPTIONAL_HEADER {

    USHORT  Magic;                          // not-so-magical number
    UCHAR   MajorLinkerVersion;             // linker version
    UCHAR   MinorLinkerVersion;
    ULONG   SizeOfCode;                     // size of .text in bytes
    ULONG   SizeOfInitializedData;          // size of .bss (and others) in bytes
    ULONG   SizeOfUninitializedData;        // size of .data,.sdata etc in bytes
    ULONG   AddressOfEntryPoint;            // RVA of entry point
    ULONG   BaseOfCode;                     // base of .text
    ULONG   BaseOfData;                     // base of .data
    ULONG   ImageBase;                      // image base VA
    ULONG   SectionAlignment;               // file section alignment
    ULONG   FileAlignment;                  // file alignment
    USHORT  MajorOperatingSystemVersion;    // Windows specific. OS version required to run image
    USHORT  MinorOperatingSystemVersion;
    USHORT  MajorImageVersion;              // version of program
    USHORT  MinorImageVersion;
    USHORT  MajorSubsystemVersion;          // Windows specific. Version of SubSystem
    USHORT  MinorSubsystemVersion;
    ULONG   Reserved1;
    ULONG   SizeOfImage;                    // size of image in bytes
    ULONG   SizeOfHeaders;                  // size of headers (and stub program) in bytes
    ULONG   CheckSum;                       // checksum
    USHORT  Subsystem;                      // Windows specific. subsystem type
    USHORT  DllCharacteristics;             // DLL properties
    ULONG   SizeOfStackReserve;             // size of stack, in bytes
    ULONG   SizeOfStackCommit;              // size of stack to commit
    ULONG   SizeOfHeapReserve;              // size of heap, in bytes
    ULONG   SizeOfHeapCommit;               // size of heap to commit
    ULONG   LoaderFlags;                    // no longer used
    ULONG   NumberOfRvaAndSizes;            // number of DataDirectory entries
    IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
} IMAGE_OPTIONAL_HEADER, *PIMAGE_OPTIONAL_HEADER;
```

First, take a look at that last member, **DataDirectory**. The constant, **IMAGE_NUMBEROF_DIRECTORY_ENTRIES** can, and has, changed through the years. This is that member that could change the size of this structure. We will look closer at that member a little later though as thats where all the fun stuff is at.

You might be interested in why this is called an "optional" header even though its clearly not optional. This is due to it being optional for COFF object files. While its not optional for executable images, it is for object files :)

magic can be one of the following:

- 0x10b: 32bit executable image
- 0x20b: 64bit executable image
- 0x107: ROM image

In the usual case, it should be **0x10b**.

A lot of the members in this structure really arent that complex.

The **subsystem** member is Windows-specific. It tells Windows what subsystem the program requires in order to execute properly. It can be one of the following values (posted here for completness only)

- 0: Unknown
- 1: Native SubSystem
- 2: GUI SubSystem
- 3: CUI SubSystem
- 5: OS/2 CUI SubSystem
- 7: POSIX CUI SubSystem
- 9: Windows CE GUI SubSystem
- 10: EFI
- 11: EFI Boot Driver
- 12: EFI Runtime Driver
- 13: EFI ROM

- 14: XBox
- 16: Boot application

The **DllCharacteristics** member contains bit flags that gives the loader information about the DLL. It follows the following format:

- **Bit 0-3**: reserved
- **Bit 4**: If set, DLL is relocatable
- **Bit 5**: If set, force code integrity checks
- **Bit 6**: If set, image is **Data Execution Prevention (DEP)** compatable
- **Bit 7**: If set, image should not be isolated
- **Bit 8**: If set, image does not use **Structured Exception Handling (SEH)**
- **Bit 9**: If set, image wont be binded
- **Bit 10**: reserved
- **Bit 11**: If set, image is a **Windows Driver Model (WDM)** driver
- **Bit 12**: reserved
- **Bit 13**: image is terminal server aware

AddressOfEntryPoint is an important one. This member contains the RVA of the entry point function of the image (for DLLs this can be null as DLLs dont need entry points.) This is what our bootloader uses to call our entry point function in our kernel.

Thats about all there is to it. You might be interested in what those other members are - for **.text**, **.data**, **.bss** etc. There is also that nasty looking **DataDirectory** member that we havnt looked at.

We will look at those members closly later on. For now, lets look at executing a program!

Executing a program

At this stage, **if all that you would like to do is execute a program**, all of the information has been provided. After loading a program, all that the loader needs to do is locate the **AddressOfEntryPoint** member from the optional header, and call that address. Remember that this is an RVA, meaning the loader needs to add this address to the **ImageBase** to obtain the linear address to the entry point function.

Here is an example:

```
//! loadedProgram is where the image was loaded to
IMAGE_DOS_HEADER* pImage = (IMAGE_DOS_HEADER*) loadedProgram;

//! go to NT HEADERS
IMAGE_NT_HEADERS* pHeaders = (IMAGE_NT_HEADERS*)(loadedProgram + pImage->e_lfanew);

//! get image base and entry point address from optional header
int base = pHeaders->OptionalHeader.ImageBase;
int entryPoint = pHeaders->OptionalHeader.AddressOfEntryPoint;

//! entry point function is at base+entryPoint
void (*entryFunction) () = (entryPoint + base);

//! call program entry point
entryFunction();
```

Thats all that is needed to execute a PE executable :)

Data Directories

Abstract

Resources, symbol tables, debugging information, import, export tables etc are accessable from that nifty **DataDirectory** member of the optional header. This member is an array of **IMAGE_DATA_DIRECTORY**'s that can be used to access other structures containing this information. **IMAGE_DATA_DIRECTORY** has the format:

```
typedef struct _IMAGE_DATA_DIRECTORY {
  DWORD VirtualAddress;         // RVA of table
  DWORD Size;                   // size of table
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;
```

Remember that **DataDirectory** is an **array** of **IMAGE_DATA_DIRECTORY**'s. Each entry in this array allows us to access the different data that we want to access.

Here are the index entries:

- 0: Export directory
- 1: Import directory
- 2: Resource directory
- 3: Exception directory
- 4: Security directory
- 5: Base relocation table
- 6: Debug directory
- 7: Description string
- 8: Machine value (MIPS GP)
- 9: TLS directory
- 10: Load configuation directory
- 14: COM+ data directory

For example, if you need to read the export table, reference **DataDirectory[0]**. If you want to read a resource, reference **DataDirectory[2].VirtualAddress**:

Each of these sections contains their own structures that are required to parse the specific data. Lets take a look at some of the more useful ones.

Reading the export table

The export table contains all functions exported from libraries or DLLs, including their function addresses within that DLL, their names, and ordinal number. The Win32 API function **GetProcAddress()** works by parsing the modules export table by ordinal number or name and returning the address from it. This is one way reading the export table can be useful.

To parse the export table, you need to first get the export directory structure. This is done by getting **DataDirectory[0]**.

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [0];
PIMAGE_EXPORT_DIRECTORY exportDirectory = (PIMAGE_EXPORT_DIRECTORY) (DataDirectory->VirtualAddress + ImageBase);
```

Remember that **VirtualAddress** in the **IMAGE_DATA_DIRECTORY** structure is an RVA, so must be added to the image base. Now **exportDirectory** points to this nice structure:

```
typedef struct _IMAGE_EXPORT_DIRECTORY {
  uint32_t Characteristics;
  uint32_t TimeDateStamp;
  uint16_t MajorVersion;
  uint16_t MinorVersion;
  uint32_t Name;
  uint32_t Base;
  uint32_t NumberOfFunctions;
  uint32_t NumberOfNames;
  uint32_t** AddressOfFunctions;
  uint32_t** AddressOfNames;
  uint16_t** AddressOfNameOrdinal;
}IMAGE_EXPORT_DIRECTORY,*PIMAGE_EXPORT_DIRECTORY;
```

This one is an easy one. **AddressOfFunctions** is an RVA that points to an array of function addresses. The function addresses, however are also RVAs. **AddressOfNames** is a pointer to a list of function names. All of these addresses are RVAs however so must be added to the image base in order to properly obtain the function name and address.

AddressOfNameOrdinal is an RVA to a list of ordinals. The ordinals, being just numbers representing the exported functions and not addresses, arent RVAs.

To properly parse the export table must be done in a loop. For example:

```
PDWORD  FunctionNameAddressArray = ((DWORD)ExportDirectory->AddressOfNames) + ((PBYTE)imageBase);
PWORD   FunctionOrdinalAddressArray = (DWORD)ExportDirectory->AddressOfNameOrdinal + (PBYTE)imageBase;
PDWORD  FunctionAddressArray = (DWORD)ExportDirectory->AddressOfFunctions + (PBYTE)imageBase;

//! search for function in exports table
for ( i = 0; i < ExportDirectory->NumberOfFunctions; i++ )
{
        LPSTR   FunctionName = FunctionNameAddressArray [i] + (PBYTE)imageBase;

        if (strcmp (FunctionName, funct) == 0) {

                WORD Ordinal = FunctionOrdinalAddressArray [i];
                DWORD FunctionAddress = FunctionAddressArray [Ordinal];
                return (PBYTE) (FunctionAddress + (PBYTE)imageBase);
        }
}
```

This can be used to impliment **GetProcAddress()** which can be useful in supporting DLLs.

Reading the import table

So... reading the export table wasnt hard enough, huh? Reading the import table isnt too hard, but is a little more involved then the export table. Ok, ok, whats the use for reading the import table? Its not so much the reading, but the **writing**. By writing entries into a programs inport table, you can allow function calls across libraries and DLLs without the need of a GetProcAddress() call. Windows performs this with delayed loaded DLLs and system DLLs.

In order to read the import table, you need to locate the import directory structure. This is at **DataDirectory[1]**:

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [1];
PIMAGE_IMPORT_DESCRIPTOR importDirectory = (PIMAGE_IMPORT_DESCRIPTOR) (DataDirectory->VirtualAddress + ImageBase);
```

It is important to note that **importDirectory** points to an **array** of descriptor entries. Each of these entries represents a module that was imported, such as an import DLL. Lets take a look at this structure:

```
typedef struct _IMAGE_IMPORT_DESCRIPTOR {
    union {
        uint32_t   Characteristics;             // 0 for terminating null import descriptor
        uint32_t   OriginalFirstThunk;          // RVA to INT
    };
    uint32_t   TimeDateStamp;                   // Time/Date of module, or other properties (see below)
    uint32_t   ForwarderChain;                  // Forwarder chain ID
    uint32_t   Name;                            // Module name
    uint32_t   FirstThunk;                      // RVA to IAT (if bound this IAT has actual addresses)
} IMAGE_IMPORT_DESCRIPTOR;
typedef IMAGE_IMPORT_DESCRIPTOR *PIMAGE_IMPORT_DESCRIPTOR;
```

Its important to note that **Name, OriginalFirstThunk** and **FirstThunk** are RVAs. This means you will need to add the addresses (these are pointers) to the image base in order to properly parse the data. **Name** is an RVA that points to the imported module name, such as **kernel32.dll**. It is null terminated.

Remember that we are working with an array of import descriptors? How can we tell how many import descriptors that is in this array? The array ends with a null IMAGE_IMPORT_DESCRIPTOR, so an easy way to loop through each entry is this:

```
IMAGE_IMPORT_DESCRIPTOR* lpImportDesc;
while (! lpImportDesc->FirstThunk) {

        //! work with lpImportDesc here
```

```
        lpImportDesc++; // move to next entry
}
```

TimeDateStamp can be either the proper time/date or one of the following values:

- 0: module not bound
- -1: image is bound. Real time/date stamp stored

ForwarderChain is only used when supporting **DLL Forward Referencing**, which allows calls across DLLs to be forwarded to other DLLs. For example, some calls in Windows **kernel32.dll** are forwarded to other DLLs.

FirstThunk points to the IAT, **OriginalFirstThunk** points to an array of structures representing all imported functions. This is the **Import Name Table (INT)**. Both of these members are RVAs.

Thunk? right, Im sure you know another structure is coming up. Lets take a look:

```
typedef struct _IMAGE_THUNK_DATA {
        union {
                uint32_t* Function;                             // address of imported function
                uint32_t  Ordinal;                              // ordinal value of function
                PIMAGE_IMPORT_BY_NAME AddressOfData;            // RVA of imported name
                DWORD ForwarderString1                          // RVA to forwarder string
        } u1;
} IMAGE_THUNK_DATA, *PIMAGE_THUNK_DATA;
```

OriginalFirstThunk are RVAs that point to an array of **IMAGE_THUNK_DATA** structures.

Ugh, yey, another structure. This one is a small one though:

```
typedef struct _IMAGE_IMPORT_BY_NAME {
        uint16_t  Hint;         // Possible ordinal number to use
        uint8_t   Name[1];      // Name of function, null terminated
} IMAGE_IMPORT_BY_NAME, *PIMAGE_IMPORT_BY_NAME;
```

Thats all there is to it. The first paramater can be 0, but it just hints the loader what ordinal number the function might be using. **Name** is an array of characters representing the name of the function.

Heres the deal: The IAT is just a list of addresses representing functions. What functions? The functions within this IMAGE_THUNK_DATA array. Look back at that IMAGE_THUNK_DATA structure and notice that its just an union representing a function name. This is the **Import Name Table (INT)**.

For example, lets say we want to get the current address of the function thats in IMAGE_THUNK_DATA[3]. Its address will be the 3rd dword in the IAT, which can be read using **IMAGE_IMPORT_DESCRIPTOR->FirstThunk**.

So, lets try to obtain the function name and address:

```
unsigned int count=0;
while (lpThunk->u1.Function) {

        //! get the function name
        char* lpFunctionName = (char*)((uint8_t*)imageBase + (uint32_t)lpThunk->u1.AddressOfData.Name);

        //! go into the IAT to get this functions address
        uint32_t* addr = (uint32_t*)((uint8_t*)imageBase + lpImportDesc->FirstThunk) + count;

        // lpFunctionName now points to the null terminated function name
        // addr now points to the address of this function

        count++;
        lpThunk++;
}
```

Image binding

This is where things get interesting. The IAT can be filled with the addresses of the imported functions either during **runtime** or **building** time. A **bounded image** is an image that has its IAT bounded to the functions during build time. An **unbounded** image is an image whose IAT is filled in by the OS loader during loading time.

If the image is bounded, you can do the following to call a function in an external DLL:

```
__declspec (dllimport) void function ();
function (); // calls myDll:function()
```

If the image is not bounded, the IAT contains junk. **It is then the responsbility of the OS loader to update the IAT** in order for the above code to work. This can be performed by reading the export table of the loaded DLL module (calling GetProcAddress(), and overwriting the IAT entry of that import function. Overwriting the IAT can be done by following the above - when you get the functions IAT entry, just overwrite it :)

This method can also be useful for installing **hooks** in DLLs and other modules.

Supporting resources

Introduction

Have you ever wondered how the Windows kernel can display an image and work with an XML configuation file without loading anything from disk? Have you ever worked with adding resources but wondered if it was possible to support them in an OS? The answer is a "yes, of course!"

Parsing resources is a bit more complex then the other directory types, however. Like the other sections, there is a base **IMAGE_RESOURCE_DIRECTORY** structure that can be obtained from the **DataDirectory** member of the optional header:

```
PIMAGE_DATA_DIRECTORY DataDirectory = &OptionalHeader->DataDirectory [2];
PIMAGE_RESOURCE_DIRECTORY resourceDirectory = (PIMAGE_RESOURCE_DIRECTORY) (DataDirectory->VirtualAddress + ImageBase);
```

Notice the pattern with how to access these sections? Oh, right, onto the new structure:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY {
    uint32_t                        Characteristics;
    uint32_t                        TimeDateStamp;
    uint16_t                        MajorVersion;
    uint16_t                        MinorVersion;
    uint16_t                        NumberOfNamedEntries;
    uint16_t                        NumberOfIdEntries;
    IMAGE_RESOURCE_DIRECTORY_ENTRY DirectoryEntries[1];
} IMAGE_RESOURCE_DIRECTORY, *PIMAGE_RESOURCE_DIRECTORY;
```

This structure doesnt have much of any interesting fields, except the last three.

If you have worked with Win32 resources, you might know that resources can be idenitified by ID or name. Two of the members in this structure will let us know the number of these entries, and the total amount of entries (NumberOfNamedEntries + NumberOfIdEntries), which is useful in looping through all of the entries. As you can probably guess, the entries are in the DirectoryEntries array. **DirectoryEntries** consists of an array of **IMAGE_RESOURCE_DIRECTORY_ENTRY** structures, which follow the format:

```
typedef struct _IMAGE_RESOURCE_DIRECTORY_ENTRY {
    union {
        struct {
            DWORD NameOffset:31;
            DWORD NameIsString:1;
        };
        DWORD   Name;
        WORD    Id;
    };
    union {
        DWORD   OffsetToData;
        struct {
            DWORD   OffsetToDirectory:31;
            DWORD   DataIsDirectory:1;
        };
    };
} IMAGE_RESOURCE_DIRECTORY_ENTRY, *PIMAGE_RESOURCE_DIRECTORY_ENTRY;
```

Alright, this is an ugly structure. This structure represents a single resource, or resource directory.

Resource directory structure

resource or resource directory? Lets stop for a moment. (*grabs a cup of coffee*) ok, it is important to know that resources are stored as a **tree**. This tree is structured like this:

- Root directory
  - Resource group 1 Directory
    - Resource 1
    - Resource 2
  - Resource group 2 Directory
    - Resource 1
    - Resource 2
  - Resource group 3 Directory
    - Resource 1
    - Resource 2
  - ...etc...

There are a number of different resource groups which let us know the type of resources are in this group. Here are the group IDs:

- 1 - Cursor
- 2 - Bitmap
- 3 - Icon
- 4 - Menu
- 5 - Dialog
- 6 - String
- 7 - Font directory
- 8 - Font
- 9 - Accelerator
- 10 - RcData
- 11 - Message table
- 16 - Version
- 17 - DlgInclude/li>
- 19 - Plug and Play
- 20 - VXD
- 21 - Animated Cursor
- 22 - Animated Icon
- 23 - HTML
- 24 - Manifest

In order to locate a resource, you will need to traverse this tree. The good news is that this isnt hard if you assume there is only 3 layers in the tree.

First, lets look at looping through all of the entries in a resource directory:

```
//! get first entry in directory
IMAGE_RESOURCE_DIRECTORY_ENTRY* lpResourceEntry = lpResourceDir->DirectoryEntries;

//! loop through all entries
int entries = lpResourceDir->NumberOfIdEntries + lpResourceDir->NumberOfNamedEntries;
while (entries-- != 0) {
```

```
        //! look for bitmap resource (id=2)
        if (lpResourceEntry->Id == 2) {
                //! see below
        }
        lpResourceEntry++;
}
```

This is simple enough, huh? The **Id** member of **IMAGE_RESOURCE_DIRECTORY_ENTRY** is used to store the group ID. If we were looking for a bitmap, it would be in the bitmap group of the root directory, so look for the entry with ID=2.

Because IMAGE_RESOURCE_DIRECTORY_ENTRY represents both resource entries and directories, how can we tell what it is? Why, the **DataIsDirectory** member of course: If this member is set, its a directory. Ah, but if its a directory, how can you read the directory? Lets take a look:

```
if (lpResourceEntry->DataIsDirectory) {
        lpResourceEntry = lpResourceEntry->OffsetToDirectory;
        lpResourceEntry += startOfResourceSection;
}
```

This one isnt to bad. If the entry is a directory, the above obtains the offset to the new directory from the **OffsetToDirectory** and adds it to .. what? the *startOfResourceSection*!? Thats right... this is an offset, but not an RVA. I know ... *Why Microsoft, Why!?*

The start of the resource section is actually the address of the first member of the IMAGE_RESOURCE_DIRECTORY_ENTRY array. So by adding this address to the offset obtained from **OffsetToDirectory** you can obtain the pointer to the **IMAGE_RESOURCE_DIRECTORY** structure for this directory. Yes, then the whole process of reading those directory entries begins :)

If you are in the process of parsing the directory for your specific resource, just loop through all of the resource entries in the directory. If the resourceEntry ID field matches that of the resource ID you are trying to find (program specific ID here), then you have found the resource data.

The resource data is stored in a ... zomg! structure! It can be obtained from the **OffsetToData** member of the directory entry structure. Simular to the **OffsetToDirectory** member, this too is an offset from the start of the resource section.

Once you obtained the pointer, you can extract the resource data. Lets take a look at that structure:

```
typedef struct _IMAGE_RESOURCE_DATA_ENTRY {
        uint32_t   OffsetToData;
        uint32_t   Size;
        uint32_t   CodePage;
        uint32_t   Reserved;
} IMAGE_RESOURCE_DATA_ENTRY, *PIMAGE_RESOURCE_DATA_ENTRY;
```

Thats it! **OffsetToData** is an RVA to the real resource data, and **Size** is the size of that data, in bytes. For example, if we were looking for a bitmap resource, **OffsetToData** would be the RVA pointing to the bitmaps **BITMAPINFOHEADER** structure, which can be handled by any bitmap loader.

Conclusion

Thats all for this chapter. There are some planned updates to including covering additional sections (debug data and COMDATS) as well.

There is no demo for this chapter - it is primarily released for anyone that is interested in the internal workings of the PE executable file format and working with it. For the main series, we might only be loading and executing the program, so all of the other information is provided for completeness only. All code provided in text for demenstration has been tested (slightly modified) to work.

In the upcoming chapters, we will be using the PE executable file format and building a loader for supporting user mode programs. After that, on to multitasking!

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Chapter 23 Home

Operating Systems Development - 8259A PIC Microcontroller

by Mike, 2007

This series is intended to demonstrate and teach operating system development from the ground up.

8259A PIC Microcontroller with all pins labled.

Introduction

Welcome! :)

This tutorial covers a very important topic: The **Programmable Interrupt Controller**. We will need to initialize this microcontroller by mapping it to our IRQ's. This will be needed when setting up interrupts, and handling interrupt requests.

This is our first controller tutorial. All of these controller tutorials go very deep in each device, while building a workable interface to handling them. Remember tha as we are in protected mode, we have nothing to guide us. One wrong move can cause unpredictable results. As we have no helping hand, we have to communicat with each controller directly. Because of this, we have emphisized hardware programming concepts all througout this series so our readers have more experience and better understanding of hardware level programming.

This tutorial puts everything we learned to the test. I will do my best to keep things simple. the **8259A Microcontroller**, Also known as the **Programmable Interrupt Controller (PIC)**.

Ready?

Get Ready

This is our first of many microcontroller programming tutorials. We will cover nearly every asset of each microcontroller as we cover them. The main series will refrence these tutorials on an as needed bases to help cover what we need these controllers for.

This tutorial is fairly complicated. We will cover the 8259A Microcontroller from both hardware and software perspectives, and understand exactally how it connects and enteracts with the PC. We will also cover every command, register, and part of this microcontroller.

History

To do - We plan on adding this section soon

Because the 8259A PIC handles hardware interrupts, we should first have a basic understanding of what interrupts are, and how they work.

Interrupts

An **Interrupt** is an external asynchronous signal requiring a need for attention by software or hardware. It allows a way of interrupting the current task so that we can execute something more important.

Not to hard. Interrupts provide a way to help trap problems, such as divide by zeros. If the processor find a problem with the currently executing code, it provides the processor alternative code to execute to fix that problem.

Other interrupts may be used to provide a way to service software as routines. These interrupts can be called by any software from within the system. This is used alot for System API's, which provide a way for ring 3 applications to execute ring 0 level routines.

Interrupts provide alot of use, espically as a way of recieving information from hardware that may change its state at asynchronous times.

Interrupt Types

There are two types of interrupts: **Software Interrupts** and **Hardware Interrupts**.

Software Interrupts

Software Interrupts are interrupts implimented and triggered in software. Normally, the processor's instruction set will provide an instruction to service software interrupts. For the x86 architectures, these are normally **INT imm**, and **INT 3**. It alsu uses **IRET** and **IRETD** instructions.

For example, here we generate an interrupt through a software instruction:

```
int     3               ; generates interrupt 3
```

These instructions may be used to generate software interrupts and execute **Interrupt Routines (IR)'s** through software.

We will not cover software interrupts here. The 8259A PIC Micrcontroller only services hardware interrupts. Software interrupts will be covered in another tutorial.

Hardware Interrupts

A hardware interrupt is an interrupt triggered by a hardware device. Normally, these are hardware devices that require attention. The hardware Interrupt handler will be required to service this hardware request.

Spurious Interrupt

This is a hardware interrupt generated by electrical interference in the interrupt line, or faulty hardware. We do NOT want this!

Interrupt Modes

There are several modes and classes of interrupts that we will need to cover. In programming the PIC, we will need to choose a mode.

Note: This section may require some knowedge of the 8259A PIC hardware pin layout. this is discussed in the next section.

Level Triggered

A **Level Triggered** interrupt is determined to happen when the **Interrupt Request (IR) line on the PIC** has current (1). A device sends a signal (Setting this line to active), and keeps it at that state until the interrupt is serviced.

Level Triggered interrupt lines may be shared by multiple interrupts if the circuit is designed to handle it.

This mode is the preferred mode because of how the lines are shared. When an IR line is active, the CPU searches through all of the devices sharing the same line until it finds what device is activating the signal. After finding the device, the CPU rechecks all of the devices again to insure there are no other devices that also need service.

A problem with this approch is, if there is an interrupt with higher priority that needs to be serviced, all other interrupts will be perminately blocked until the other interrupts are serviced. After all, only one line can be active at a given time.

Edge Triggered

Edged Triggered interrupts are determnined to happen when the **Interrupt Request (IR) line on the PIC** has current (1). A device sends a signal (Setting this line to active) through a single pulse, and returns the line to its previous state.

Edged Triggered interrupt lines may be shared by multiple interrupts if the circuit is designed to handle it.

If the pulse is too short to be detected, then it will not be detected.

As these are only pulses of current that signals interrupt requests, Edged triggered mode does not have the same problems that Level triggered does with shared IRQ lines.

Of course, we still run into the possibility of an interrupt being missed, as it is just a single pulse of current being sent through the IRQ line. This has caused early computer lockups of the CPU.

However, through recent times, these lockups have decreased through time.

Hybrid

Both of these modes have their pros and cons. Alot of systems impliment a hybrid of both of them. More specifically, Most systems check for both Edge triggered and Level triggered interrupts on the **Non Maskable Interrupt (NMI)** pin on the CPU. **The purpose of this is that the NMI pin is used to signal major problems with the system that can cause big problems, or entire system malfunctions, possibly hardware damage.**

The Non Maskable Interrupt is just that -- It cannot be disabled or masked off by any device. This insures, along with having a hybrid setup, that if the NMI pin is set, the system can die peacefully without big problems.

Message Signaled

These types of hardware interrupts do not use a physical interrupt line. Instead, they rely on another medium, such as the system bus, to send messages over.

These types of interrupts cause the device to only send a pulse of current over the medium, similar to edge triggered interrupts.

These types of systems may use a special interrupt line on its control bus indicating a message signaled interrupt number. As these numbers are sent over the medium as a series of bits, they do not have the limitations of the other interrupt types, which are limited to a single interrupt line. As such, they can manage as much interrupts as the underlaying system allows. These types of interrupts also support sharing of interrupt vectors.

PCI Express uses these types of interrupts alot.

Thats all

Okay, alot of info here ;) The 8259A only has support for Level triggered and Edge triggered interrupts. Because of this, Those should be your primary focus when working with the 8259A Microcontrollers.

Interrupt Vector Table

The **Interrupt Vector Table (IVT)** is a list of **Interrupt Vectors**. **There are 256 Interrupts in the IVT.**

Interrupt Routines (IR)

An **Interrupt Routine (IR)** is a special function used to handle an **Interrupt Request (IRQ)**.

When the processor executes an **interrupt instruction**, such as **INT**, it executes the **Interrupt Routine (IR)** at that location within the **Interrupt Vector Table (IVT)**.

This means, it simply executes a routine that we define. Not to hard, huh? This special routine determins the **Interrupt Function** to execute normally based off of the value in the AX register. This allows us to define multiple functions in an interrupt call. Such as, the DOS INT 21h function 0x4c00.

Remember: Executing an interrupt simply executes an interrupt routine that you created. For example, the instruction **INT 2** will execute the IR at index 2 in the IVT. Cool?

IVT Map

The IVT is located in the first 1024 bytes of physical memory, from addresses 0x0 through 0x3FF. Each entry inside of the IVT is 4 bytes, in the following format:

- **Byte 0:** Offset Low Address of the **Interrupt Routine (IR)**
- **Byte 1:** Offset High Address of the IR
- **Byte 2:** Segment Low Address of the IR
- **Byte 3:** Segment High Address of the IR

Notice that each entry in the IVT simply contain the address of the IR to call. This allows us to create a simple function anywhere in memory (Our IR). As long as the IVT containes the addresses of our functions, everything will work fine.

Okay, Lets take a look at the IVT. The first few interrupts are reserved, and stay the same.

x86 Interrupt Vector Table (IVT)		
Base Address	Interrupt Number	Description
0x000	0	Divide by 0
0x004	1	Single step (Debugger)
0x008	2	Non Maskable Interrupt (NMI) Pin
0x00C	3	Breakpoint (Debugger)
0x010	4	Overflow
0x014	5	Bounds check
0x018	6	Undefined Operation Code (OPCode) instruction
0x01C	7	No coprocessor
0x020	8	Double Fault
0x024	9	Coprocessor Segment Overrun
0x028	10	Invalid Task State Segment (TSS)
0x02C	11	Segment Not Present
0x030	12	Stack Segment Overrun
0x034	13	General Protection Fault (GPF)
0x038	14	Page Fault
0x03C	15	Unassigned
0x040	16	Coprocessor error
0x044	17	Alignment Check (486+ Only)
0x048	18	Machine Check (Pentium/586+ Only)
0x05C	19-31	Reserved exceptions
0x068 - 0x3FF	32-255	Interrupts free for software use

Not to hard. Each of these interrupts are located at a base address within the IVT.

Interrupt Handling in Protected Mode (PMode)

Protected Mode requires each IVT entry to point to an interrupt routine (IR) defined within an **Interrupt Descriptor Table (IDT)**. The IDT will be explained furthe in another tutorial, as it is not directly related to this tutorial.

The IDT is an array of **Interrupt Descriptors**, that describe the base address of the Interrupt Routine (IR) to execute, that contains extra information about it's protection level, segment information, etc. PMode uses a Global Descriptor Table (GDT) that defines the memory map that is being used. Most of the interrupt routines will be inside of a code descriptor, mapped by the GDT. This is why the IDT is required in PMode.

Do not worry if you do not understand this right now. For now, just think of it as an array of 256 function pointers, mapped exactly like that of the IVT (It normally is, anyways.)

Hardware Interrupts

There 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 o 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

You do not need to worry to much about each device just yet. The 8259A Pins will be described in detail within the next section. The Interrupt Numbers listed in th table are the default DOS **interrupt requests (IRQ)** to execute when these events trigger.

In most cases, we will need to recreate a new interrupt table. As such, most operating systems need to remap the interrupts the PIC's use to insure they call the proper IRQ within their IVT. This is done for us by the BIOS for the real mode IVT. We will cover how to do this later in this tutorial as well.

8259 Programmable Interrupt Controller

The 8259 Microcontroller familiy is a set of **Programmable Interrupt Controller (PIC) Integrated Circuits (ICs)**. Look back again at **Tutorial 7**... Under the **Processor Architecture** section, **Notice that the processor has it's own internal PIC Microcontroller.** This is very important to note.

Do to limitations in the circuit design, **a PIC only supports 8 IRQ's**. This is a big limitation. As additional devices were created, IBM quickly realized that this limitation is very bad. Because of this, **Most motherboards contain a secondary (Slave) PIC microcontroller to work with the primary PIC inside the**

processor. Today, this is very common. A single PIC can be "cascaded" (capable of working with) another PIC. This makes it possible to support more IRQ's with additional PICs.

The More PIC's supported, the more IRQ's can be handled. They can be cascaded to support up to 64 IRQ's. Cool?

Remember: Most computers have 2 PIC's, 1 inside the processor, and 1 on the motherboard. Some systems may not have this.
Remember: Each PIC can only support up to 8 IRQ's.
Remember: Each PIC can communicate with each other, allowing up to 64 IRQ's depending on the number of PIC's.

Not to hard :)

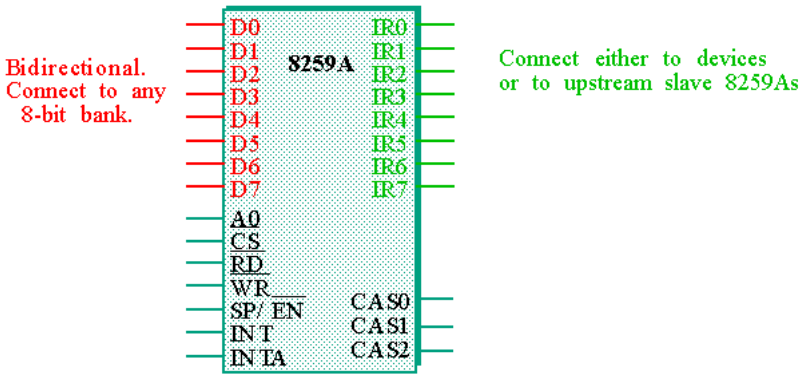
8259 Hardware

Understanding how microcontrollers work at the hardware level will help in understanding how the software side of things work. **Remember that the PIC's are only used during a hardware interrupt.**

8259A Microcontroller

At the top of this tutorial, there is an image of an actual 8259 **Dual Inline Package (DIP)**, with all of the electronic pins labled. To make things more understandable, we are going to represent the controller using a simpler graphic. **The only pins these graphics do not display that the 8259 has are GND (Ground) and Vcc (Input Voltage).** You can see these pins labled in the picture on the top of this tutorial.

Lets first look at what we are going to be programming:



Thats it--The 8259A Programmable Interrupt Controller.

Each of the lines in the above image displays each of the controllers electronic pins. These electronic pins are the connections between the controller and the rest c the system.

This is the chip that we will need to program in order to handle IRQ's within an operating system. Let's look at this closer at each pin. I **bolded** the important pins.

- **WR Pin:** This pin connects to a write strobe signal (One of 8 on a Pentium)
- **RD Pin:** This connects to the **IOCR (Input Output Control Routine)** signal.
- **INT Pin: Connects to the INTR pin on the microprocessor.**
- **INTA Pin: Connects to the INTA pin on the microprocessor.**
- **A0 Pin: Selects different Command WORDS**
- **CS Pin:** Enables the chip for programming and control.
- **SP/EN Pin:** Slave program (SP) / Enable Buffer (EN).
  - Slave Program (1=Master, 0=Slave)
  - Enable Buffer (Controls data bus transievers when in buffered mode)
- **CAS0, CAS1, CAS2 Pins: Used to output from master to slave PIC controllers in cascaded systems.**
- **D0 - D7 Pins: 8 bit Data connector pins.**

There are a couple of important pins here. Pins D0-D7 provide a way for an external device to communicate with the PIC. This is like a small data bus--It provides way to send data over to the PIC, like...An interrupt number, perhaps?

Remember that we can connect PIC's together. This allows us to provide support for up to 64 IR numbers. In other words--64 hardware interrupts. CAS0, CAS1, and CAS2 pins provide a way to send signals between these PIC's.

Look at the INT and INTA pins. Remember from the **Processor Perspective** section that the processors' own INT and INTA pins connect to these pins on the PIC. Remember that, when about to execute an interrupt, the processor clears the Interrupt (IF) and Trap flags (TF) from the FLAGS register, which disables the INTR pin. **The PIC's INT pin connects to the processors' INTR pin.**

This means that the processor, essentally, disables the PIC's INT pin when executing an interrupt.

With this, the pins IR0-IR7 can be streamed to other PIC's. These 8 pins represent the 8 bit interrupt number to be executed. Notice that this, as being an 8 bit value, provides a way to allow up to 256 hardware interrupts. these lines provide a way to send the interrupt number to another PIC controller, so that controller could handle it instead.

The important thing to note is that **We can combine multiple PIC's to support more interrupt routine numbers.** The IR lines connect to another PIC's data lines to transfer data over. As there are only 8 lines (8 bits), we can only connect up to 8 PIC's together, providing support for up to 64 interrupt numbers.

Okay... Alot of stuff here, huh? We have described how the processor connects to the primary PIC, and how the PIC's can combine with other PIC's to create a chai of PIC's.

This is great, but completely useless. How does an interrupt execute through hardware? What makes this controller "programmable"? How can we program the PIC to work for our needs?

Programming the PIC revolves around the use of sending **Command Bytes** through the 8 bit data line that the PIC's have. This 8 bit command byte follows specifi formats that describe what the PIC is to do. We will need to know these commands in order to program the PICs. We w8ill cover this later.

Lets take a closer look at how the PIC works. This will help in better understanding of the 8259A pins, and how interrupt signals are sent.

8259A Connections

Note: This section may require some knowledge in Digital Logic Electronics.

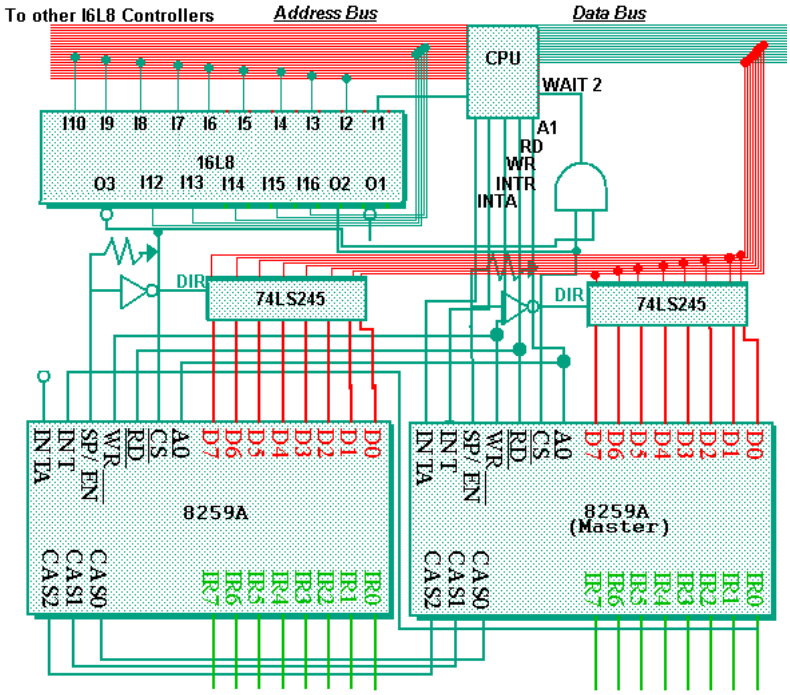
Okay... So far we have looked at the 8259A PIC pins. Lets try to look at these pins from another perspective, and see what it looks like within a typical computer.

Connecting the PICs to the processor

First, remember when I said that most computers today have 2 8259A PICs? This is only half true. Remember from the **Processor Architceture** that the primary PIC is integrated into the processor. There is a reason for this, as you will soon see.

To make things simple, lets imagine the system we are on actually have 2 PIC controllers, both directly on the motherboard (None of them are integrated with the processor.)

Looking at this graphically, this is what we might see:



Okay... There is ALOT going on here. This displays a part of the IO Subsystem and ISA bus, and how the 8259A controllers connect to the system bus through the common 16L8.

Do not worry at all if you do not understand this, as digital logic electronics is not a prerequisite for this series ;) This image is also missing more details as well. Nontheless, it displays the basic links and connections between the components.

Looking at the above image, there are a few important notes.

Notice how the slave controller connects to the primary controller.

Notice that only the primary PIC needs direct connection with the processor. Because of this tight integration, modern computers useually have the primary PIC integrated directly inside of the processor to eleminiate this dependency.

Also notice how the CAS0-CAS2 pins directly connect to the second PIC. This allows the primary PIC to send commands to the secondary PIC.

And, as we all know, the IR lines connect to other controllers that control that line. For example, **Hardware Interrupt 0 represents a timer interrupt. The 8254 Programmable Interval Timer (PIT) Controller will send a signal through the IR0 line to the primary PIC, as it is directly connected to it.** This signal will either be a current that stays active until the interrupt has been serviced, or may be a single pulse that is held for a certain time. **We can control what we want the PIC to watch for. This is described in more detail later.**

So...There you have it :) The infamous 8259A PIC Microcontroller.

How hardware interrupts execute

On the underside of all microprocessors contain connectors. These can be flat, or in the form of pins, that connects to the motherboard. Two of these pins are the INTR and NMI pins. With this, there is another pin for acknowledges the completion of the interrupt - INTA.

Software interrupts are handled differently then hardware interrupts. Both of these types of interrupts are inside of the **Interrupt Vector Table** located at address 0 through 0x3ff in memory.

Remember: Only hardware interrupts are handled through the Programmable Interrupt Controller.

The interrupt is generated

When a device controller needs to generate an interrupt, it needs to signal the PIC somehow. Lets say, for purposes of discussion, that this device is the timer, which uses interrupt line 0.

1. The timer controller signals the PIC by activating the IR0 line. This changes its state from a 0 (No power) to a 1 (Power is going through the line.)
2. The PIC sets the bit representing the IRQ inside of the **Interrupt Request Register (IRR)**. In this example, bit 0 will be set to 1.
3. The PIC examins the **Interrupt Mask Register (IMR)** to see if the interrupt can be serviced.
    - If the interrupt can be serviced, the PIC determins if there are any higher priority interrupts waiting to be serviced. If there is, the interrupt request is ignored until the higher priority interrupts are serviced.
    - If the interrupt can be serviced, and there are no higher priority interrupts, the PIC continues to the next step.
4. The PIC signals the processor through the INTA pin to inform the processor an interrupt has been fired.

The processor now knows that an interrupt has been fired.

The processor acnowledges the interrupt

1. The CPU completes execution of the current instruction.
2. The CPU examins the **Interrupt Flag (IF)** within **RFLAGS**.
    - If IF is set, the CPU acnowledges the interrupt request through the INTR pin back to the PIC.
    - If IF is cleared, the interrupt request is ignored.
3. The PIC recieves the acnowledgment signal through INTR.
4. The PIC places the interrupt vector number into the D0-D7 pins.
    - This interrupt vector number is abtained from the **Initialization Control Word (ICW) 2** during initialization of the PIC. We will cover this later.
5. The PIC also places the IRQ number into D0-D7
6. The PIC sets the correct bit inside of the **In Service Register (ISR)**. In this case, it is bit 0. This indicates that Interrupt 0 is currently being serviced.

Now the processor has the IRQ number and the interrupt vector number to execute.

Interruption

1. The processor interrupts the current process. It pushes EFLAGS, CS, and EIP on the stack.
2. The processor uses the interrupt vector number (given by the PIC).
    - In real mode, the CPU offsets into the IVT. In Protected Mode, The Processor offsets into the IDT.
    - Real Mode:
        - The CPU offsets into the correct entry into the IVT
        - The CPU loads the base address of the interrupt to call into CS:IP
        - The interrupt takes control.
    - Protected Mode:
        - The CPU uses the loaded IDT to offset into
        - The selecter feild of the gate descriptor is loaded into the CS segment selector.
        - The offset feild of the gate descriptor is loaded into EIP.
        - If paging is enabled, this address is translated from a linear address to a physical address.
        - Now, the CPU will perform architecture specific security checks on the current state.
        - The interrupt routine can now take control from gate descriptor + CS:EIP.

The Interrupt Service Routine

Now the ISR is executing to handle the hardware interrupt. It can perform whatever action needed to service the specific device. For example, reading or writing data to/from the device, reading status registers, sending commands, et al.

During this time, all interrupts are masked out by the **Interrupt Mask Register (IMR)**. In other words, this disables all hardware interrupts until a request has been made to end the interrupt. this requires an **End of Interrupt (EOI)** command to be sent to the PIC.

After the EOI signal has been sent to the PIC through the Primary PIC's **Command Register**, The PIC cleares the approprate bit in the **In Service Register (IRR)**, and is now ready to service new interrupts.

the interrupt service routine then performs a **IRETD** instruction, popping EFLAGS, CS, and EIP registers, which were pushed by the processor when the interrupt was fired.

This transfers control back to the initil task.

8259A Registers

The 8259A has several internal registers, simular to the processor.

Command Register

This is a write only register that is used to send commands to the microcontroller. There are alot of different commands that you can send. Some commands are used to read from other registers, while other command are used to initialize and sending data, such as End of Interrupt (EOI). We will cover these commands late

Status register

This is a read only register that can be accessed to determin the status of the PIC.

Interrupt Request Register (IRR)

This register specifies which interrupts are pending acknowledgment.

Note: This register is internal, and cannot be accessed directly.

Interrupt Request Register (IRR)		
Bit Number	IRQ Number (Primary controller)	IRQ Number (Slave controller)
0	IRQ0	IRQ8
1	IRQ1	IRQ9
2	IRQ2	IRQ10
3	IRQ3	IRQ11
4	IRQ4	IRQ12
5	IRQ5	IRQ13
6	IRQ6	IRQ14
7	IRQ7	IRQ15

If a bit is set, the interrupt has been signaled by a device, and the PIC has signaled the CPU, but is awaiting acknowledgment from the CPU to go ahead with the interrupt.

In-Sevice Register (ISR)

This register specifies which interrupts have already been acknowledged, but are awaiting for the **End of Interrupt (EOI)** signal. The EOI signal is very important as it determins the end of an interrupt.

Note: We will need to send the EOI signal upon completion of the interrupt to let the 8259A acknowledge the interrupt. Not doing so will result in undefined behavior or malfunction. More on this later.

Note: This register is internal, and cannot be accessed directly.

In Service Register (ISR)		
Bit Number	IRQ Number (Primary controller)	IRQ Number (Slave controller)
0	IRQ0	IRQ8
1	IRQ1	IRQ9
2	IRQ2	IRQ10
3	IRQ3	IRQ11
4	IRQ4	IRQ12
5	IRQ5	IRQ13
6	IRQ6	IRQ14
7	IRQ7	IRQ15

If a bit is set, the current IRQ has been acknowledged by the CPU to go ahead and begin executing. The PIC uses this register to determin what IRQ is currently being executed.

Interrupt Mask Register (IMR)

This specifies what interrupts are to be ignored, and not acknowledged. this allows us to focus on executing certain, more important interrupts before executing th interrupts specified in this register.

This is an 8 bit register, where each bit determins if an interrupt is disabled or not. If the bit is 0, it is enabled. If it is a 1, the interrupt device is disabled.

Interrupt Mask Register (IMR)		
Bit Number	IRQ Number (Primary controller)	IRQ Number (Slave controller)
0	IRQ0	IRQ8
1	IRQ1	IRQ9
2	IRQ2	IRQ10
3	IRQ3	IRQ11
4	IRQ4	IRQ12
5	IRQ5	IRQ13
6	IRQ6	IRQ14
7	IRQ7	IRQ15

This is an important register, as it allows us to enable and disable interrupts from certain devices. Each of these IRQ's represent the device listed in the **x86 Hardware Interrupts** table shown above.

For example, lets say we want to enable COM1 (Serial Port 1). Looking at the x86 Hardware Interrupt Table, we can see that this is mapped to IRQ 4. So, in order to enable COM1 interrupts, all we need to do is set the IRQ4 bit for the primary PIC's Interrupt Mask Register. This register is mapped to the software port number 0x21 (We will cover this later.) So, all we need to do is set the bit by writing to this port location.

```
in      al, 0x21                ; read in the primary PIC Interrupt Mask Register (IMR)
and     al, 0xEF                ; 0xEF => 11101111b. This sets the IRQ4 bit (Bit 5) in AL
out     0x21, al                ; write the value back into IMR
```

Too cool for school B)

When a hardware interrupt occurs, **The 8259A Masks out all other interrupts until it recieves an End of Interrupt (EOI) signal.** We will need to send the EOI upon completion of the interrupt. We will look at this later.

8259A Software Port Mappings

Like all hardware controllers, the BIOS POST maps each controller to use a specific region of software ports. Because of this, in order to communicate with the PIC controllers, we need to use software ports.

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

Notice the Primary PIC's Interrupt Mask Register is mapped to Port 0x21. We have seen this before, havn't we?

The **Command Register** and **Status Register** are to different registers that share the same port number. The command register is write only, while the status register is read only. This is an important difference, as the PIC determins what register to access depending on weather the write or read lines are set.

We will need to be able to write to these ports to communicate with individual device registers and control the PICs. Lets now take a look at the commands for the PIC.

8259A Commands

Setting up the PIC is farily complex. It is done through a series of **Command Wowrds**, which are a bit pattern that containes various of states used for initialization and operation. This might seem a little complex, but it is not to hard.

Because of this, lets first look at how to initialize the PIC controllers for our use, followed by operating and controlling the PICs.

Initialization Control Words (ICW)

The purpose of initializing the PIC is to remap the PIC's IRQ numbers to our own. this insures the proper IRQ is generated when a hardware interrupt happens.

In order to initialize the PIC, we must send a command byte (Known as an **Initialization Control Word (ICW)**) to the primary PIC Command Register. **This is ICW 1**.

There can be up to 4 Initialization Control Words. These are not required, but are often needed. Lets take a look at them.

Note: If there are multiple PICs in the system that are to be cascaded with each other, we must send the ICW's to both of the PICs!

ICW 1

This is the primary control word used to initialize the PIC. this is a 7 bit value that must be put in the primary PIC command register. This is the format:

Initialization Control Word (ICW) 1		
Bit Number	Value	Description
0	IC4	If set(1), the PIC expects to recieve IC4 during initialization.
1	SNGL	If set(1), only one PIC in system. If cleared, PIC is cascaded with slave PICs, and ICW3 must be sent to controller.
2	ADI	If set (1), CALL address interval is 4, else 8. This is useually ignored by x86, and is default to 0
3	LTIM	If set (1), Operate in Level Triggered Mode. If Not set (0), Operate in Edge Triggered Mode
4	1	Initialization bit. Set 1 if PIC is to be initialized

5	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0
6	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0
7	0	MCS-80/85: Interrupt Vector Address. x86 Architecture: Must be 0

As you can see, there is alot going on here. We have seen some of these before. This is not as hard as it seems, as most of these bits are not used on the x86 platform.

To initialize the primary PIC, all we need to do is create the initil ICW and set the appropraite bits. So, lessee...

- **Bit 0** - Set to 1 so we can sent ICW 4
- **Bit 1** - PIC cascading bit. x86 architectures have 2 PICs, so we need the primary PIC cascaded with the slave. Keep it 0
- **Bit 2** - CALL address interval. Ignored by x86 and kept at 8, so keep it 0
- **Bit 3** - Edge triggered/Level triggered mode bit. By default, we are in edge triggered, so leave it 0
- **Bit 4** - Initialization bit. Set to 1
- **Bits 5...7** - Unused on x86, set to 0.

Looking at the above, the final bit pattern becomes **00010001**, or 0x11. So, to initialize the PIC, send 0x11 to the primary PIC controller register, mapped to port 0x20...

```
; Setup to initialize the primary PIC. Send ICW 1
mov     al, 0x11
out     0x20, al

; Remember that we have 2 PICs. Because we are cascading with this second PIC, send ICW 1 to second PIC command register
out     0xA0, al        ; slave PIC command register
```

Because we have enabled cascading, we need to send ICW 3 to the controller as well. Also, because we have set bit 0, we must also send ICW 4. More on those later. For now, lets take a look at ICW 2.

ICW 2

This control word is used to map the base address of the IVT of which the PIC are to use. **This is important!**

Initialization Control Word (ICW) 2		
Bit Number	Value	Description
0-2	A8/A9/A10	Address bits A8-A10 for IVT when in MCS-80/85 mode.
3-7	A11(T3)/A12(T4)/A13(T5)/A14(T6)/A15(T7)	Address bits A11-A15 for IVT when in MCS-80/85 mode. **In 80x86 mode, specifies the interrupt vector address.** May be set to 0 in x86 mode.

During initialization, we need to send ICW 2 to the PICs to tell them where the base address of the IRQ's to use. If an ICW1 was sent to the PICs (With the initialization bit set), you must send ICW2 next. **Not doing so can result in undefined results.** Most likley the incorrect interrupt handler will be executed.

Unlike ICW 1, which is placed into the PIC's data registers, ICW 2 is sent to the data Registers, as software ports 0x21 for the primary PIC, and port 0xA1 for the secondary PIC. (Please see the **8259A Software Port Map** table for a complete listing of PIC software ports).

Okay, so assuming we have just sent an ICW 1 to both PICs (Please see the above section), lets send an ICW 2 to both PICs. This will map a base IRQ address to both PICs.

This is very simple, but we must be careful at where we map the PICs to. Remember that the first 31 interrupts (0x0-0x1F) are reserved (Please see the above **x8 Interrupt Vector Table (IVT)** table). As such, we have to insure we do not use any of these IRQ numbers.

Instead, lets map them to IRQs 32-47, right after these reserved interrupts. the first 8 IRQ's are handled by the primary PIC, so we map the primary PIC to the base address of 0x20 (32 decimal), and the secondary PIC at 0x28 (40 decimal). Remember there are 8 IRQ's for each PIC.

```
; send ICW 2 to primary PIC
mov     al, 0x20                ; Primary PIC handled IRQ 0..7. IRQ 0 is now mapped to interrupt number 0x20
out     0x21, al

; send ICW 2 to secondary controller
mov     al, 0x28                ; Secondary PIC handles IRQ's 8..15. IRQ 8 is now mapped to use interrupt 0x28
out     0xA1, al
```

That is simple, huh? Onto the next one!

ICW 3

This is an important command word. It is used to let the PICs know what IRQ lines to use when communicating with each other.

ICW 3 Command Word for Primary PIC

Initialization Control Word (ICW) 3 - Primary PIC		
Bit Number	Value	Description
0-7	S0-S7	Specifies what Interrupt Request (IRQ) is connected to slave PIC

ICW 3 Command Word for Secondary PIC

Initialization Control Word (ICW) 3 - Secondary PIC		
Bit Number	Value	Description
0-2	ID0	IRQ number the master PIC uses to connect to (**In binary notation**)
3-7	0	Reserved, must be 0

We must send an ICW 3 whenever we enable cascading within ICW 1. this allows us to set which IRQ to use to communicate with each other. Remember that the 8259A Microcontroller relies on the IR0-IR7 pins to connect to other PIC devices. With this, it uses the CAS0-CAS2 pins to communicate with each other.

We need to let each PIC know about each other and how they are connected. We do this by sending the ICW 3 to both PICs containing which IRQ line to use for both the master and associated PICs.

Remember: The 80x86 architecture uses IRQ line 2 to connect the master PIC to the slave PIC.

Knowing this, and remembering that we need to write this to the data registers for both PICs, we need to follow the formats shown above.

Note that, in the ICW 3 for the primary PIC, **Each bit represents an interrupt request.** That is...

IRQ Lines for ICW 2 (Primary PIC)	
Bit Number	IRQ Line
0	IR0
1	IR1
2	IR2
3	IR3
4	IR4
5	IR5
6	IR6
7	IR7

Notice that IRQ 2 is Bit 2 within ICW 3. So, in order to set IRQ 2, we need to set bit 2 (Which is at 0100 binary, or 0x4).

Here is an example of sending ICW 3 to the primary PIC:

```
; Send ICW 3 to primary PIC
mov     al, 0x4         ; 0x4 = 0100 Second bit (IR Line 2)
out     0x21, al        ; write to data register of primary PIC
```

To send this to the secondary PIC, we must remember that we must send this in **binary notation**. Please refer to the table above. Note that only Bits 0...2 are used to represent the IRQ line. By using binary notation, we can refer to the 8 IRQ lines to choose from:

IRQ Lines for ICW 2 (Secondary PIC)	
Binary	IRQ Line
000	IR0
001	IR1
010	IR2
011	IR3
100	IR4
101	IR5
110	IR6
111	IR7

Simple enough. Notice that this just follows a binary<->decimal conversition in the above table.

Because we are connected by IRQ line 2, we need to use bit 1 (Shown above).

Here is a complete example, that sends a ICW 2 to both primary and secondary PIC controllers:

```
; Send ICW 3 to primary PIC
mov     al, 0x4         ; 0x04 => 0100, second bit (IR line 2)
out     0x21, al        ; write to data register of primary PIC

; Send ICW 3 to secondary PIC
mov     al, 0x2         ; 010=> IR line 2
out     0xA1, al        ; write to data register of secondary PIC
```

Thats all there is to it ;)

Okay, so now both PICs are connected to use IR line 2 to communicate with each other. We have also set a base interrupt number for both PICs to use.

This is great, but we are not done yet. Remember that, when building up ICW 1, **if bit 0 is set, the PIC will be expecting us to send it ICW 4.** As such, we need to send ICW 4, the final ICW, to the PICs.

ICW 4

Yey! This is the final initialization control word. This controls how everything is to operate.

Initialization Control Word (ICW) 4		
Bit Number	Value	Description
0	uPM	If set (1), it is in 80x86 mode. Cleared if MCS-80/86 mode
1	AEOI	If set, on the last interrupt acknowledge pulse, controller automatically performs End of Interrupt (EOI) operation
2	M/S	Only use if BUF is set. If set (1), selects buffer master. Cleared if buffer slave.
3	BUF	If set, controller operates in buffered mode
4	SFNM	Special Fully Nested Mode. Used in systems with a large amount of cascaded controllers.
5-7	0	Reserved, must be 0

This is a pretty powerful function. Bits 5..7 are always 0, so lets focus on the other bits and peices (pun entended ;))

The PIC was originally designed to be a generic microcontroller, even before the 80x86 existed. As such, it containes alot of different operation modes designed for different systems. one of these modes is the **Special Fully Nested Mode**.

The x86 family does not support this mode, so you can saftley set bit 4 to 0.

Bit 3 is used for buffered mode. For now, set this to 0. We will cover modes of operation later. Bit 2 is only used when bit 3 is set, so set this to 0. With this, Bit 1 is rairly used either.

As such, we only need to set bit 0, which enables the PIC for 80x86 mode.

Simple enough. So, to send ICW 4, all we need to do is this:

```
        mov     al, 1           ; bit 0 enables 80x86 mode

        ; send ICW 4 to both primary and secondary PICs
        out     0x21, al
        out     0xA1, al
```

This is probably the easiest code snipplet in this tutorial. Brace it while it lasts! :)

Initializing the PIC - Putting it together

Believe it or not, but we have already went over this. In initilizing the PIC, all we need to do is send the correct ICW's to the PIC.

Lets put everything from the previous section together to initialize the PIC for better understanding of how everything is put together:

```
;*****************************************************************
; Map the 8259A PIC to use interrupts 32-47 within our interrupt table
;*****************************************************************

%define ICW_1 0x11                              ; 00010001 binary. Enables initialization mode and we are sending ICW 4

%define PIC_1_CTRL 0x20                         ; Primary PIC control register
%define PIC_2_CTRL 0xA0                         ; Secondary PIC control register

%define PIC_1_DATA 0x21                         ; Primary PIC data register
%define PIC_2_DATA 0xA1                         ; Secondary PIC data register

%define IRQ_0   0x20                            ; IRQs 0-7 mapped to use interrupts 0x20-0x27
%define IRQ_8   0x28                            ; IRQs 8-15 mapped to use interrupts 0x28-0x36

MapPIC:

; Send ICW 1 - Begin initialization -------------------------

        ; Setup to initialize the primary PIC. Send ICW 1

        mov     al, ICW_1
        out     PIC_1_CTRL, al

; Send ICW 2 - Map IRQ base interrupt numbers ---------------

        ; Remember that we have 2 PICs. Because we are cascading with this second PIC, send ICW 1 to second PIC command register

        out     PIC_2_CTRL, al

        ; send ICW 2 to primary PIC

        mov     al, IRQ_0
        out     PIC_1_DATA, al

        ; send ICW 2 to secondary controller

        mov     al, IRQ_8
        out     PIC_2_DATA, al

; Send ICW 3 - Set the IR line to connect both PICs ---------

        ; Send ICW 3 to primary PIC

        mov     al, 0x4                 ; 0x04 => 0100, second bit (IR line 2)
        out     PIC_1_DATA, al          ; write to data register of primary PIC

        ; Send ICW 3 to secondary PIC

        mov     al, 0x2                 ; 010=> IR line 2
        out     PIC_2_DATA, al          ; write to data register of secondary PIC

; Send ICW 4 - Set x86 mode ---------------------------------

        mov     al, 1                   ; bit 0 enables 80x86 mode

        ; send ICW 4 to both primary and secondary PICs

        out     PIC_1_DATA, al
        out     PIC_2_DATA, al

; All done. Null out the data registers

        mov     al, 0
        out     PIC_1_DATA, al
        out     PIC_2_DATA, al
```

That was not that hard, was it? We covered everything in this code.

Now the PIC is initialized. Whenever an hardware interrupt accors, it will call our interrupts 32 - 47 that we have previously defined somewhere within the Interrup Vector Table (IVT). This allows us to track hardware interrupts. Cool, huh?

Operation Command Words (OCW)

Yippee! Now that the ugly initialization stuff is out of the way, we can finally focus on standard controlling and operation of the PIC. This is done by writing and reading from various registers through **Operation Control Words (OCW)'s**.

OCW 1

OCW 1 represents the value inside of the **Interrupt Mask register (IMR)**. To abtain the current OCW 1, all you need to do is read from the IMR.

Remember that the IMR is mapped to the same port that the status register is at. Because the status register is read only, the PIC can determin what register to access based off if this is a read or write operation.

We have looked at the IMR register above when we covered the PIC registers.

OCW 2

This is the primary control word used to control the PIC. Lets take a look...

Operation Command Word (OCW) 2		
Bit Number	Value	Description
0-2	L0/L1/L2	Interrupt level upon which the controller must react
3-4	0	Reserved, must be 0
5	EOI	End of Interrupt (EOI) request
6	SL	Selection
7	R	Rotation option

Okay then! Bits 0-2 represents the interrupt level for the current interrupt. Bits 3-4 are reserved. Bits 5-7 are the interesting bits. Lets take a look at each combination for these bits.

OCW2 Commands			
R Bit	SL Bit	EOI Bit	Description
0	0	0	Rotate in Automatic EOI mode (CLEAR)
0	0	1	Non specific EOI command
0	1	0	No operation
0	1	1	Specific EOI command
1	0	0	Rotate in Automatic EOI mode (SET)
1	0	1	Rotate on non specific EOI
1	1	0	Set priority command
1	1	1	Rotate on specific EOI

Okay...This table, in its current state, is confusing, don't you think? Alot of the above commands are fairly advanced. Lets take a look at what we can do.

Sending End of Interrupt (EOI)

As you know, when a hardware interrupt triggers, all other interrupts are masked off inside of the **Interrupt Mask Register** until an EOI signal is sent to the primary controller. This means, we must send an EOI to insure all hardware interrupts are enabled at the end of our **Interrupt Routine (IR)**.

Looking at the above table, we can send a non specific EOI command to signal EOI to the controller. Because the EOI bit is bit 5 within the OCW 2, all we need to d is set bit 5 (100000 binary = 0x20):

```
; send EOI to primary PIC

mov     al, 0x20        ; set bit 4 of OCW 2
out     0x20, al        ; write to primary PIC command register
```

Conclusion

The PIC is a complex microcontroller to program. We have covered alot in this tutorial, and it is only going to get harder!

I hope I explained everything well. The OS Development Series primary tutorial series will put everything inside of this tutorial where it belongs. It will be the glue between setting up interrupts, interrupt handling, and hardware interrupts ;)

I plan on expanding on this tutorial to provide more content, and to insure we describe every detail reguarding the 8259A microcontrollers.

This is a side tutorial that the main series will use. As such, there is no demo for this tutorial. I might decide to make one, however, to being everything together. However, this will require us to cover the IDT, and interrupt handling in detail here, which is not relivant to the PIC. Not directly relivant, anyways.

I hope this tutorial will help answer alot of questions one may have when programming the PIC, and understanding what is really going on behind the hood.

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Operating Systems Development - 8253 Programmable Interval Timer

by Mike, 2008

This series is intended to demonstrate and teach operating system development from the ground up.

8253 PIT Microcontroller with all pins labled

Please note: This tutorial may require some knowledge of hardware interrupt handling and the 8259 Programmable Interrupt Controller (PIC). Please see this tutorial for information on them.

Introduction

Welcome! :)

This tutorial will cover everything you ever wanted to know about system timing and programming the **Intel 8253 Programmable Interval Timer (PIT)**.

The 8253 PIT has had a long history, and has played an important part in nearly every x86 PC. It is the "System clock", and is responsible for many very important functions within the PC. This erm.. "chip" is no longer distributed as an independent chip (as a **Dual Inline Package (DIP)** to be more precise), but rather integrated into the motherboards southbridge chipset.

Everything about the 8253 still remains, however. Because of this, its input and output facilities, hardware, and the way we program the 8253 remains the same. Because there is not of any difference (besides speed) between this and the older DIPs, we will be looking at the older 8253 DIP to help keeping things simple.

The picture at the beginning of this tutorial displays what we will be looking at and programming.

Lets have some fun ;)

Programmable Interval Timers

A **Programmable Interval Timer (PIT)** is a counter which triggers an interrupts when they reach their programmed count. The **8253** and **8254** microcontrollers are PITs avialble for the i86 architectures used as timer for i86-compatable systems.

These PICs include three timers that are used for different putposes. The first timer is useually used as the **System Clock**. Timer 2 was used for RAM refreshing, and timer 3 is connected to the PC speaker. We will see all of the connections a little later, so we won't go into much detail now.

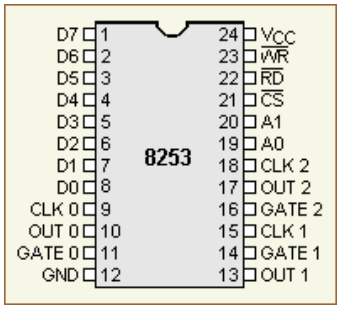
Instead, lets take a closer look at one of these famous PITs... The 8253 Microcontroller.

8253 Hardware

Before looking at the software side of things, it will be helpful to learn more about what we are actually programming. Because of this, we will look at the 8253 hardware first, and learn how it works and is connected to the rest of the PC. We will also be looking at internal registers, pin layout, command words, and more that will be needed for the software side of things.

8253 Hardware: Description

The 8253 PIT has a simple interface, and is not that hard to program.



Yep--There is the chip that we will program.

Here is the complete pin layout. We will be refrencing these pins throughout this tutorial, so it is important to know what they are.

- **D0...D7:** 8 bit data lines. This is connected to the data bus so we can read and send commands.
- **CLK 0, CLK 1, CLK 2:** Clock input pins. There are 3 pins for 3 separate counters
- **OUT 0, OUT 1, OUT 2:** Output data line. There are 3 pins for 3 separate counters
- **GATE 0, GATE 1, GATE 2:**Gate data line. There are 3 pins for 3 separate counters
- **GND:** Ground
- **Vcc:** Input voltage
- **WR:** Write enable. When this line is active, lets the 8253 that we are writing data
- **RD:** Read enable. When this line is active, lets the 8253 that we are reading data
- **CS:** Chip select signal
- **A0, A1:** Address lines. Used to determin what register we are accessing.

Not to bad. There are a couple of important pins here that we need to look at.

The **D0-D7** pins connect to the **systems data bus**. These pins carry our data when we are sending or reading data to the controller.

Vcc and **GND** complete the circuit (Voltage input, Ground output.)

The **WR** pin tells the controller that we are writing (expect input on the data pins.) When the signal in this pin is "low", we are currently sending data. The **RD** pin is very simular in this manner, but it tells the controller we are reading data instead. The **CS** pin is a special pin that determins what the controller should do with the **RD** and **WR** pins. If the CS pin is "low", the controller will respond to the RD and WR pins. If CS is not, they are ignored. **WR** and **RD** connect to the **Systems Control Bus**. The **CS** pin connects to the systems **Address Bus** for port i/o operations.

The **A0** and **A1** pins are connected to the **Systems Address Bus**, and are used to determin what register we are accessing. These, in conjunction with **WR** and **RD** allows the controller to determin if are reading to or from a register.

Notice that there are three groups of the CLK, OUT, and GATE pins. Yes, there is indeed a reason for this: **The 8253/8254 microcontrollers contain 3 independent timers.**

Lets look closer on how this works...

8253 Hardware: Counters

The 8253 consists of three counters: Counter 0, Counter 1, and Counter 2. Each counter has 2 input pins: **CLK (Clock Input)** and **GATE**, and one pin for output--**OUT**.

As there are three counters, they are used for different purposes within the system. Each counter are 16 bit down counters.

Typical computers connect the first timer's **OUT** pin to a **Programmable Interrupt Controller (PIC)** to generate an interrupt for every clock tick. This is useually used as the **System Timer**. The second counter was used for generating a timing signal to the Memory Controller to refresh DRAM memory. The third counter is used to generate tones to the PC speaker.

As you can probably guess, the PIT uses the OUT pins to signal these devices when its counter reaches 0. When the PIT's counter reaches 0, it simply wraps around and starts again.

CLK is the clock input for the timer. It may be used with the **GATE** pin depending on the current mode of operation. The following table describes the operation depending on if the current in GATE is low, rising, or high.

GATE Input pin operations			
Mode	Low or going low	Rising	High
0	Disables Counting	-	Enables Counting
1	-	Initiates Counting and resets OUT after next CLK	-
2	Disables Counting, Sets OUT to high	Reloads counter and initiates counting	Enables Counting
3	Disables Counting, Sets OUT to high	Initiates Counting	Enables Counting
4	Disables Counting	-	Enables Counting
5	-	Initiates Counting	-

The 8253 Counters are also known as Channels. Knowing that the 8253/8254 PITs contain three channels, lets look at each of them more closer...

Channel 0

Channel 0 is connected to the **8259 PIC** to generate an **Interrupt Request (IRQ)**. The PITs **OUT** pin connects to the PIC's **IR0** pin. Typically the BIOS configures this channel with a count of 65536, which gives an output frequency of 18.2065 Hz. This fires IRQ 0 every 54.9254 ms.

The is the primary timer used on almost all x86 machines. the clock rate (Signalled through Counter 0's **CLK** pin) is at 1193181.6666... Hz, one third of the NTSC subcarrier frequency. This was required do to backward compatability with the older CGA PC's.

Channel 0 is typically programmed in most systems to act as the **System Clock**. This is made possible do to channel 0's **OUT** pin indirectly connecting to the **PIC's IR0** line. Depending on the mode that we set it in, we can set the timer to a good frequency, and have it enable the PIC's IR0 line at a constant rate. Afterwords, reseting itself and starting over again. Because the PIC is used to handle **hardware interrupts**, we will need to first reprogram the **PIC**.

Because it is connected to the interrupt with the lowest number (IR0), it also has the highest priority over all other hardware interrupts.

Its lowest frequency rate is normally used for computers running the old DOS systems (Are there any left?) at about 18.2 Hz. Its highest frequency rate is a little over a megahertz.

In real mode operating systems, the BIOS normally increments the number of times IRQ0 is fired to 0000:046C, which can be read by any running program.

Channel 1

Many video cards and the BIOS may reprogram the second channel for their own uses. This channel was originally used for generating a timing pulse signal to signal the memory controller to refresh the DRAM memory. In modern times, this is no longer needed as the refresh is done by the memory controller. Because of this, there is no guarantee at what devices may use this counter.

Channel 2

This channel is connected to the **PC Speaker** to generate sounds. the **PC speaker** is normally meant to produce a square wave with two levels of output. However, it is possible to go between the two true defined sound square levels. This is called **Pulse-Width Modulation (PWM)**.

We can set up this channel by programming it for mode 3, and setting a frequency rate for the tone.

We can also program the PC Speaker directly. Looking back at Tutorial 7, we can see that the PC speaker is mapped to port 0x61. This port defines how the speaker will operate:

- **Bit 0:** If set (1), the state of the speaker follows bit 1
- **Bit 1:** If set (1), the speaker uses the PIT, If not set (0), the speaker disables its connection to the PIT

If bit 0 is set, the rest of the byte contains a pattern of bits representing the frequency of the tone. We can generate up to 8 bit sounds from the speaker, which is kind of cool, but tricky to do.

You should also notice that we can disable devices from using the PIT. At startup, the BIOS configures the speaker to use the PIT channel 2, running in mode 3. It is recommended to keep the speaker to use the PIT do to timing problems that you will run into.

Here is an example, for completeness sake:

```
; disables the speaker, and stop using channel 2
mov     dx, 0x61
out     dx, 0

; generates tone from speaker
out     dx, 11111101b
```

Conclusion

Once a counter is set up, it will remain that way until it is changed by another control word.

There is some cool things we can do with these counters, huh? Because channel 1 is not used anymore, we cannot assume it is safe and use it ourselves. Because of this, it is recommended to stick to channel's 0 and 2.

We can use channel 0 to fire off our interrupt handler. Our interrupt handler can increment a counter used by our kernel. This special little counter variable, plays a very important role in the system: The System Timer. We will see all of this soon, don't worry ;)

Allright...So, we looked at pin configurations, and the three timers used by different devices for..err...timing. Whats next?

When programming these timers, we have to initialize them. Remember that each channel supports 6 different modes. Some of these modes are very useful. Other modes are not. Lets take a look at each of these modes for better understanding of them. Please note that this gets a little detailed, but I am sure you already knew or were expecting that :)

8253 Channel Modes

Remember that each counter can be programmed in 1 of 6 modes. This is done by sending an **Initialization Control Word (ICW)** to the controller. We will look at the format of this command word later. For now, lets look at each mode.

Mode 0: Interrupt on Terminal Count

In this mode, the counter will be programmed to an initil **COUNT** value and afterwords counts down at a rate to the input clock frequency (The **CLK** signal). When **COUNT** is equal to 0, and after the **Control Word** is written, the counter enables its **OUT** pin (by setting its line high) to signal the device it is connected to. Counting starts one clock cycle after the **COUNT** is programmed. The OUT line remains high until the counter is reloaded with a new value or the same value or until another control word is written to the controller.

What this mode basically does is allow us to set a timer that counts down to 0. After which, we will need to reload a new count number to it, or a new control word to re-initialize the counter.

Mode 1: Hardware Triggered One-Shot

In this mode, the counter is programmed to give an output pulse every certain number of clock pusles. The **OUT** line is set to high as soon as a **Control Word** is written. After **COUNT** is written, the counter waits until the rising edge of the **GATE** input. If the trigger occurs during the pulse output, the 8253 will be retriggered again. One clock cycle after the rising edge of **GATE** is detected, **OUT** will become and remain low until **COUNT** reaches 0. **OUT** will then be set high until the next trigger and wait again until the rising edge of the **GATE** input is detected.

Mode 2: Rate Generator

This mode configures the counter to be a **"divide by n"** counter, which is commonly used to generate a real-time system clock.

The counter is programmed to an initial **COUNT** value. Counting starts at the next clock cycle. **OUT** remains high until **COUNT** reaches 1. Afterwords, **OUT** will be set low for one clock pulse. **OUT** is then set back high, and **COUNT** is reset back to its initial value. This process repeats until a new control word is sent to the controller.

The time between the high pulses depends on the current value in **COUNT**, and is calculated using the following formula:

```
COUNT = input (Hz) / Frequency of output
```

COUNT never reaches 0. It only ranges from n to 1, where n is the initil COUNT value.

Okay, lets stop for a moment. Remember that Counter 0 is connected to the PIC? **Counter 0's OUT line indirectly connects to the PIC's IR0 line**. Knowing that, when the IR0 line is low, the PIC will call the IRQ 0 handler defined by us.

If we set the counter to Mode 2, we can set up the timer to fire off our interrupt at a constant rate. All we need to do is determine what the COUNT value should be based off of the above formula. This is used very often in setting up the **System Timer** for the operating system. After all, IRQ 0 is now being called for every clock tick at a frequency rate that we defined.

Yes, Mode 2 is an important mode, indeed.

Mode 3: Square Wave Generator

This mode is quite similar to Mode 2. However, **OUT** will be high for half of the period, and low for the other half. If **COUNT** is odd, **OUT** will be high for (n+1)/2 counts. If **COUNT** is even, **OUT** will be low for (n-1)/2 counts.

Everything else is the same from Mode 2. We will need to use the formula from Mode 2 to set up the initial **COUNT** value.

If the speaker is configured to use the PIT, the channel that it uses typically should be set to use this mode.

Mode 4: Software Triggered Strobe

The counter is programmed to an initial **COUNT** value. Counting starts at the next clock cycle. **OUT** remains high until **COUNT** reaches 0. The counter will then set **OUT** low for one clock cycle. Afterwords, it resets **OUT** to high again.

Mode 5: Hardware Triggered Strobe

The counter is programmed to an initial **COUNT** value. **OUT** remains high until the controller detects the rising edge of the **GATE** input. When this happens, the counting starts. When **COUNT** reaches 0, **OUT** goes low for one clock cycle. Afterwords, **OUT** is set high again. This cycle repeats when the controller detects the next rising edge of **GATE**.

8253 Registers

The 8253 contains a few registers that we can access. Most of these registers are very simular to each other, so I will just put them in the same table for clarity. This table displays each register and their functionality when the corresponding lines on the 8253 are active. **Notice how the RD and WR lines determine the read and write operation. Also notice how the A0 and A1 lines determin what register we are accessing.**

Looking at the port table from Tutorial 7, we can see that the **System Timer** is mapped by the BIOS to use ports 0x40-0x4F. Each port address is a byte in size.

8253 PIT Internal Registers						
Register Name	Port Address	RD line	WR line	A0 line	A1 line	Function

Counter 0	0x40	1	0	0	0	Load Counter 0
		0	1	0	0	Reads Counter 0
Counter 1	0x41	1	0	0	1	Load Counter 1
		0	1	0	1	Reads Counter 1
Counter 2	0x42	1	0	1	0	Load Counter 2
		0	1	1	0	Reads Counter 2
Control Word	0x43	1	0	1	1	Write Control Word
NA		0	1	1	1	No Operation

All other port addresses from 0x44-0x4f are undefined.

The system will activate the correct lines depending on the operation we are performing. When setting the counter registers, we need to first let the controller know how we are going to load it. This is done by first setting up the control word. Lets take a look closer at these registers...

Counter Registers

Each counter register holds the **COUNT** value used by the PIT to count down from. They are all 16-bit registers. When writing or reading from these registers, you must first send a control word to the PIT. You might wonder why we cannot just do it directly. There is a reason for this, and it has to do with the size of the data. The PIT only has 8 data lines (Pins D0-D7). However, the counter registers are all 16 bits, not 8.

Because of this, how does the PIT know what data you are writing to its counter register? How does it know what byte within the counter registers 16 bits are you setting? ***It doesn't.*** Sending a command word allows you to let the PIT know to expect incoming data, and what to do with it. We will look at that next.

Control Word Register

THIS will be important to us.

This is an important register used to determin and set the operation modes for the controller. This is accessed by enabling the RD, A0, and A1 lines. This register can only be written to, not read from.

The control word register uses a simple format. At first I was thinking of using a table here, but it may be easier in a list format so here it is:

- **Bit 0: (BCP)** Binary Counter
  - **0:** Binary
  - **1:** Binary Coded Decimal (BCD)
- **Bit 1-3: (M0, M1, M2)** Operating Mode. See above sections for a description of each.
  - **000:** Mode 0: Interrupt or Terminal Count
  - **001:** Mode 1: Programmable one-shot
  - **010:** Mode 2: Rate Generator
  - **011:** Mode 3: Square Wave Generator
  - **100:** Mode 4: Software Triggered Strobe
  - **101:** Mode 5: Hardware Triggered Strobe
  - **110:** Undefined; Don't use
  - **111:** Undefined; Don't use
- **Bits 4-5: (RL0, RL1)** Read/Load Mode. We are going to read or send data to a counter register
  - **00:** Counter value is latched into an internal control register at the time of the I/O write operation.
  - **01:** Read or Load Least Significant Byte (LSB) only
  - **10:** Read or Load Most Significant Byte (MSB) only
  - **11:** Read or Load LSB first then MSB
- **Bits 6-7: (SC0-SC1)** Select Counter. See above sections for a description of each.
  - **00:** Counter 0
  - **01:** Counter 1
  - **10:** Counter 2
  - **11:** Illegal value

Allright, then! bit of stuff going on here, don't you think? So...All we need to do is build up the control word by writing to the control word register to initialize the controller, right? Of course! Sort of...

Basically, we want to initialize a **counter** for a specific purpose. So, we have to build up the control word to set up the counter, the counters counting mode and operating mode. THEN, we initialize the counter itself. Remember that, once initialized, the counter can carry on itself (depending on its mode), so we only need to do this once.

Lets give an example and put everything together, shall we?

Lets pretend we already have a PIC initialized and interrupt 0 handler. We want to set up a timer to fire off IRQ 0 every 100Hz (once every 10 milliseconds). We know that channel 0 of the PIT is connected to the PIC's IR0 line, so we can program channel 0 to do this.

```
        ; COUNT = input hz / frequency

        mov     dx, 1193180 / 100       ; 100hz, or 10 milliseconds
```

```
        ; FIRST send the command word to the PIT. Sets binary counting,
        ; Mode 3, Read or Load LSB first then MSB, Channel 0

        mov     al, 110110b
        out     0x43, al

        ; Now we can write to channel 0. Because we set the "Load LSB first then MSB" bit, that is
        ; the way we send it

        mov     ax, dx
        out     0x40, al        ;LSB
        xchg    ah, al
        out     0x40, al        ;MSB
```

Notice how we set up the control word first, THEN write to the counter 0 register.

Is that it!?? Yep. The above will program counter 0 to fire IRQ0 every 10 milliseconds.

Conclusion

The 8253 and 8254 PIT's are very usefull little chips. They can be used in alot of different devices, and used for alot of different purposes.

For our needs, we can both have a nice signaled output via the PC speaker and a system timer that is a very important aspect on all modern system software. We have even looked at another way of working with the speaker directly, and even disable its connection to the PIT, which has advantages and disadvantags. It is up to the system software designer to decide and outweight the pros and cons of the different and unlimited possibilities when developing their system.

We have looked at the PIT in depth, pin configuations, and their connections on a standard x86-based PC motherboard. The PIT itself is useually integrated with the motherboard's southbridge in modern computers.

I an planning on adding more to the chip itself, and describe exactally how it connects in a typical computer system. Mabey even breaking it apart?

Tutorial 16: Kernel: Timing and Exception Handling puts everything from the 8259A PIC and this tutorial together. It impliments and creates interfaces for both devices. Mabey even a little bit more...

Welcome back to kernel Land!

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Operating Systems Development Series

Operating Systems Development - Graphics 1

by Mike, 2010

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

Welcome!

Wait, what? Graphics already? Thats right, we will start developing an ultra-cool GUI for the OS! :) Okay, not really, but its a start in that direction.

This chapter is the first of a miniseries of chapters covering graphics programming. I plan to cover Vesa VBE, Video BIOS, and direct hardware programming for the VGA and, possibly, some SVGA concepts. I also plan on covering graphics concepts and rendering, including 2d vector rendering and images. Who knows; mabye a little 3d a little later.

Excited? A lot of cool material coming up in this mini series spinoff of the OS Development Series! However, before we can dive into the wonderful world of computer graphics, we have to set up a ground rule. There are a lot of ways that we can work with computer graphics, and a lot of directions that we can take. Computer graphics are a complicated topic: It cannot be covered in one chapter. Well, it can. It would just be one .. very, very long chapter.

Because of this, I decided to do this in stages. The first chapter covers working with graphics in real or v86 modes. We use the system BIOS interrupts and cover basic graphics concepts. The second chapter we will dive into Video BIOS Extensions (VBE) and Super VGA. The third chapter, will be the first chapter of a smaller miniseries covering direct hardware programming of the graphics pipeline: VGA and mabye some Super VGA topics.

So for this chapter, lets get started with working with real mode graphics using the real mode Video BIOS...

Basic Concepts

Abstract

Computer Graphics (CG) does not need an introduction. It has revolutionized the computer, animation, and video game industries. The field of computer graphics encompasses the development, creation, and continuation of the ability of producing graphical effects on computer displays. From 1D graphics, 2D, 3D, and even 4D graphics simulation software.

History

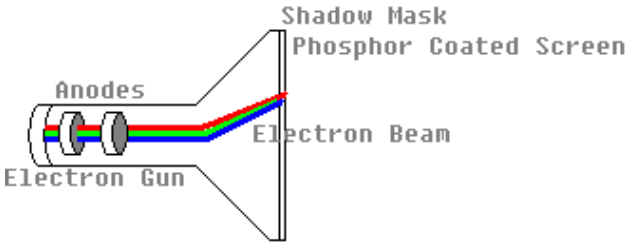
The computer graphics industry started to emerge from early projects like the **Whirlwind** in the 1960's. The Whirlwind was the first computer that used video display output and helped introduce the **Cathode Ray Tube (CRT)** technology. Whirlwind eventually led to the development of the SAGE (Air Force Semi Automatic Ground Environment) computer system. The ealiest known version of the CRT was created by Ferdinand Braun in 1897 known as the *Braun tube*.

The **Special Interest Group on GRAPHics and Interactive Techniques (SIGGRAPH)**, is governed by the **Association for Computing Machinery (ACM) SIGGRAPH** group. Originally started in 1969 by Andy van Dam, the group hosts the SIGGRAPH conferences around the world. These conferences are attended by thousands of professionals from companies from the engineering, graphics, motion picture, and video game industries.

As graphics hardware advanced the ability of creating more powerful graphics designs emerged. As other display technologies emerges, such as **Liquid Crystal Display (LCD)**, The use of the CRT technology started to decline.

Video Display Terminals (VDT), also known as a **Video Display unit (VDU)** are early display terminals.

Cathode ray tube (CRT)



Abstract

A CRT is a vacuum tube which consists of electron guns and a phosphor target. The entire front area of the tube is scanned repetitibely in a pattern called a **raster**. The image is produced by changing the intensity of the three electron beams: one for red, green, and blue color components at a given point on display. These electron beams first travel through a **Shadow Mask** layer before hitting the phosphor coated screen.

Problems

CRT monitors can emit a small amount of X-Ray radiation. Also, do to the constant rescanning of the display, at low **refresh rates** (below 60Hz) flicker may be seen. CRTs may also contain some toxic phosphore. Because of this the United States Environmental Protection Agency (EPA) created a rule that CRTs must be bought to a proper recycling facility. Finally, do to the CRT containing a vaccum of glass, if the outer glass is damaged, the CRT may implode. This may cause the glass to shatter outward at dangerous speeds. Modern CRTs have certain measures in place to prevent the shattering of the CRT.

It is possible to control the frequency of the CRT using software. At higher frequency rates, it is possible to make the CRT operate faster then its intended use increasing the possibility of imploding the CRT. Because of this, it is very important to be careful when working with the **CRT Controller (CRTC)**. Modern CRTs have protections in place to prevent this, however.

VGA

Abstract

The Video Graphics Array (VGA) is an anolog computer display standard marketed in 1987 by IBM. It is called an "Array" because it was originally developed as a single chip, replacing dozens of logic chips in a Industry Standard Architecture (ISA) board that the MDA, CGA, and EGA used. Because this was all on a single ISA board, it was very easy to connect it to the motherboard.

The VGA consists of the video buffer, video DAC, CRT Controller, Sequencer unit, Graphics Controller, and an Attribute Controller. We will cover all of these components in more detail in later chapters.

Video Buffer

The **Video Buffer** is a segment of memory mapped as Video Memory. We can change what region of memory is mapped to video memory. At startup, the BIOS maps it to 0xA0000., which means that video memory is mapped to 0xA0000. (Remember the Real Mode Address Map from Tutorial 7?) We will cover memory mapping a little later in this chapter in more detail.

Video DAC

The **Video Digital to Analog Converter (DAC)** contains the color palette that is used to convert the video data into an analog video signal that is sent to the display. This signal indicates the red, green, and blue intensities in analog form. We will go into more detail later, so don't worry if you do not understand this yet.

CRT Controller

This controller generates horizontal and vertical synchronization signal timings, addressing for the video buffer, cursor and underline timings. We will go into more detail later when we cover the VGA hardware.

Sequencer

The Sequencer generates basic memory timings for video memory and the character clock for controlling regenerative buffer fetches. It allows the system to access memory during active display intervals. Once more, we will not cover this in detail yet.

Graphics Controller

This is the interface between video memory and the **attribute controller**, and between video memory and the CPU. During active display times, memory data is sent from the video buffer (Video Memory) and sent to the Attribute Controller. In Graphics Modes, this data is converted from parallel to a serial bit plane data before being sent. In text modes, Just the parallel data is sent.

Don't worry if you do not understand these yet. I do not plan on going into much detail here. We will cover everything in detail later when we talk about developing a video driver. For now, just remember that: The Graphics Controller refreshes the display from the parallel data from video memory. This is automatic based on the active display times. This simply means, that By writing to video memory (Default mapped to 0xA0000) we effectivly write to video display, depending on the current mode. This is important when printing characters.

Remember that it is possible to change the address range used by the Graphics Cotroller. When initializing, the BIOS does just this to map video memory to 0xA0000.

Video Modes

A "Video Mode" is a specification of display. That is, it describes how Video Memory is refrenced, and how this data is displayed by the video adapter.

The VGA supports two types of modes: APA Graphics, and Text.

APA Graphics

All Points Addressable (APA) is a display mode, that, on a video monitor, dot matrix, or any device that consists of a pixel array, where every cell can be refrenced individually. In the case of video display, where every cell represents a "pixel", where every pixel can be manipulated directly. Because of this, almost all graphic modes use this method. By modifying this pixel buffer, we effectivly modify individual pixels on screen.

Pixel

A "Pixel" is the smallest unit that can be represented on a display. On a display, it represents the smallest unit of color. That is, basically, a single dot. The size of each pixel depends heavily on the current resolution and video mode.

Text Modes

A Text Mode is a display mode where the content on the screen is internally represented in terms of characters rather then pixels, as with APA.

A Video Controller implimenting text mode uses two buffers: A character map representing the pixels for each individual character to be displayed, and a buffer that represents what characters are in each cell. By changing the character map buffer, we effectivly change the characters themselves, allowing us to create a new character set. By changing the Screen Buffer, which represents what characters are in each cell, we effectivly change what characters are displayed on screen. Some text modes also allow attributes, which may provide a character color, or even blinking, underlined, inversed, brightened, etc.

MDA, CGA, EGA

Remember that VGA is based off of MDA, CGA, and EGA. VGA also supports alot of the modes these adapters do. Understanding these modes will help in better understanding VGA.

MDA

Back before I was born (Seriously :)) in 1981, IBM developed a standard video display card for the PC. They were the Monochrome Display Adapter (MDA), and Monochrome Display and Printer Adapter (MDPA).

The MDA did not have any graphics mode of any kind. It only had a single text mode, (Mode 7) which could display 80 columns by 25 lines of high resolution text characters.

This display adapter was a common standard used in older PC's.

CGA

In 1981, IBM also developed the Color Graphics Adapter (CGA), coinsidered the first color display standard for PC's.

The CGA only supported a Color Palette of 16 colors, because it was limited to 4 bytes per pixel.

CGA supported two text modes and two graphics modes, including:

- 40x25 characters (16 color) text mode
- 18x25 characters (16 color) text mode
- 320x200 pixels (4 colors) graphics modes
- 640x200 pixels (Monochrome) graphics mode

It is possible to treak the display adapter in creating and discovering new, "undocumented" video modes. More on this later.

EGA

Introduced in 1984 by IBM, The Enhanced Graphics Adapter (EGA) produced a display of 16 colors at a resolution up to 640x350 pixels.

Remember that the VGA adapters are backward compatible, simular to the 80x86 microprocessor family. Because of this, and to insure backward compatibility, the BIOS starts up in Mode 7 (Originally from the MDA), which supports 80 columns, by 25 lines. This is important to us, because this is the mode we are in!

Video Memory

Memory Mapped I/O (MMIO)

If you know what Memory Mapped I/O is, you can skip this part.

The processor can work with reading from RAM and ROM devices. In applications programming, this is something you never see. This is made possible with MMIO devices. **Memory Mapped I/O allows a hardware device to map its own RAM or ROM into your processors physical address space.** This allows the processor to be able to access hardware RAM or ROM in different ways by just using a pointer to that location in the address space. This is made possible because MMIO devices uses the same physical address and data bus that the processor and system memory uses.

It is important to remember, however, that Memory Mapped I/O is a mapping to the physical address space of the processor, not actual computer memory. In some architectures, it is possible to *bank switch*, or provide a method to switch between either using the MMIO device mapping or the system memory "hidden" behind it, while on others it is not. What this means for us is that we cannot access the actual system memory addresses that are "hidden" by the MMIO device. For example, CMOS RAM memory is mapped into the physical address space at address 0x400. This is different then main system memory; accessing 0x400 with a pointer will access the CMOS RAM memory always do to MMIO. It is not possible to access this location in system memory in the i86 architecture.

MMIO devices allows us to have more control over the hardware - it allows high resolution video displays with limited system memory, it allows us to obtain information from a device that is kept current by a battery (CMOS RAM) that would have altherwise been lost if in system memory. Another example of an MMIO device is the system BIOS ROM itself. MMIO is what allows the processor to execute the BIOS from ROM as it is mapped to the systems physical address space. Cool, huh?

You might be wondering what this has to do with graphics. Video memory is RAM that is mapped into the physical address space. Video memory is managed by the video display device which uses MMIO to do this. **How MMIO memory is managed and worked with is up to the device**; it is not always nice and linear. Different graphics modes require different ways that you have to work with this memory, so understanding that it is an MMIO device is important.

An interesting fact about MMIO address space regions is that, with paging they can be mapped to any virtual address and accessed from that address. This means you can map, for example, video memory, to any virtual address you want and access video memory using that virtual address. This, of course, has to do with the way pages are mapped to frames in the physical address space.

Also remember that MMIO memory is not in system memory. Computer system memory does not need to be greater then the size of the MMIO address that you are trying to access. For example, if your system only has 2GB of system memory, you can still access the MMIO device if it has RAM mapped to the physical address space at 0xFC000000 without error.

See this text right here? Thats right, me; I am in your computer... residing in **Video RAM (VRAM).** VRAM is Video Memory, also known as the **video framebuffer**. It containes all of the pixels that you see before you, and more.

Standard VGA

Video memory is stored inside of the video device; usually a video card or onboard video adapter. Standard VGA cards have 256 KB of VRAM. It is not uncommon however to see SVGA+ cards to have much more video memory

however. After all, they have to be able to store all of the pixels in the high resolution video modes somehow, right?

Remember the memory map from chapter 7? We can see the Standard VGA memory resides in **0x000A0000 - 0x000BFFFF**. 0xBFFFF - 0xA0000 = 0xA0000, which is 655360 bytes, or 640 KB.

It is important to remember that video memory is mapped in the PCs address space at this location. What this means is by writing here, you are writing to video memory that is located in the video adapter. This is a form of **Memory Mapped I/O**.

When accessing video memory, you typically access it using a "window" into the real video RAM. This is typically:

- 0xA0000 - EGA/VGA graphics modes (64 KB)
- 0xB0000 - Monochrome text mode (32 KB)
- 0xB8000 - Color text mode and CGA (32 KB)

Because different modes uses different address mappings, it is possible to combine a monochome display adapter and color adapter on the same machine. This allows a computer with a dual monitor setup to be able to run without issues. Of course, this is just standard VGA.

Super VGA

Super VGA and other display adapters typically do things differently. It is not uncommon to see a Super VGA or higher resolution display adapter to have VRAM mapped to a high address range. While they will usually support the Standard VGA memory mapped range, they can use other memory ranges as well to help with high resolution video modes or to provide additional functionality. For example, my NVideo GeForce 7600 GT has 4 memory ranges that it can use: 0xA0000 - 0x000BFFFFF (Look familier?), 0xFC000000 - 0xFCFFFFFF, 0xD0000000 - 0xDFFFFFFF, and 0xFD000000 - 0xFDFFFFFF. This can be different on your system.

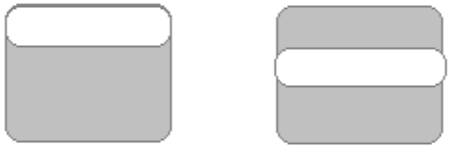
Linear Frame Buffer (LFB)

If it is possible to map the entire video memory of the current display into the physical address space, it is possible to set it up to act like a linear frame buffer. A linear frame buffer is just a packed-pixel frame buffer that allows you to be able to read or write to it in linear fashion. For example, buffer[0] is the first element of the buffer, buffer[1] is the second - there is nothing special. Well, actually there is. Standard VGA does not support LFB modes. Remember Mode 0x13 above? That is the only Standard VGA video mode that creates the effect of a linear frame buffer.

This might be a little confusing. After all, how "else" can you read or write to video memory if its not in linear fashion? This has to do with Standard VGA being a **planer** device. We will talk about that after the next section,

Bank Switching

Super VGA and higher resolution video modes can also provide a way of using a "window" into the full video memory that is on the adapter. For example, notice above that, for graphics modes, we are limited to a 64KB region between **0xA0000 - 0xB0000**. If this was a "window", and we can "move" this 64K window around, we can access a much larger video memory area. For example:

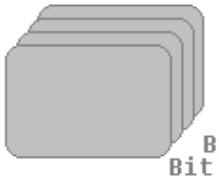


```
unsigned char* vidmem = 0xa0000;

vidmem[0] = 0; //! writes to topleft of screen
movwindow (1); //! moves window (see second picture)
vidmem[0] = 0; //! writes to topleft of screen + size of window
vidmem[0] always points to the topleft of the window, not screen
```

This is known as **Bank Switching**. A "Bank" is a window into the larger video memory. The size of the window is typically 64K do to standard VGA only having a graphics region of 64K.

Planer Memory

Okay, it gets a little tricky here. Standard VGA modes operate in planer memory mode. This is the VGAs native memory model.




```
4 planes, each sharing the same 64K window.
Each pixel's bits share the same location but
different plane.
    unsigned char* vmem = 0xa0000;

    vmem[0] = pixel_bit 0
    set_plane(1)
    vmem[0] = pixel_bit 1
    set_plane(2)
    vmem[0] = pixel_bit 2
    set_plane(3)
    vmem[0] = pixel_bit 3
```

The above is an example of Mode 12h's planer memory format. Mode 12h has 4 bits per pixel. To draw a pixel, you have to set or unset the bit in the plane. To better understand this, imagine you have a 64k block of video memory. Imagine it as a flat sheet of paper and put three more behind it. Each sheet of paper is a 64k "plane" that shares this same 64k area of memory. Each plane holds a little bit of information about the pixel that it is used for.

Dont worry to much about understanding planer memory and how it works, we wont be needing it in this chapter. It will be important when we cover VGA and Mode 12h in more detail however. Because we are using Mode 0x13, which hides the details of working with planer memory, we wont need it now.

Odd / Even Memory Addressing

Odd / Even Memory Addressing uses the Planer Memory model and **is used in all text modes**. All even addresses work with planes 0 or 2 and odd addresses work with planes 1 or 3. For example:

```
Memory Address | Plane   | Offset in plane
---------------------------------------
0              | Plane 0 | Offset 0
1              | Plane 1 | Offset 0
2              | Plane 0 | Offset 2
3              | Plane 1 | Offset 2
```

Remember what it is like writing to video memory in text modes?

```
unsigned char* vmem = 0xb8000;
                    vmem [0] = 'a'; // plane 0 [character plane] offset 0
                    vmem [1] = 0x7; // plane 1 [attribute plane] offset 0
                    vmem [2] = 'b'; // plane 0 [character plane] offset 2
                    vmem [3] = 0x7; // plane 1 [attribute plane] offset 2
```

In text modes, plane 0 is used to store character codes and plane 1 stores the attribute bytes. Plane 2 stores the font data. If you overwrite plane 2 when writing to video memory, you will overwrite the font installed at boot time by the BIOS. This means that, **if you write over plane 2 in graphics mode, and go back to text mode, the BIOS text output routines will not work as expected as the font data is corrupt.**

If you would like to go back to text mode, you will either need to store your own font or backup the default font and write it back to plane 2 before using any text output routines again.

Because we are not using the planer memory model in this chapter, we wont be using the Odd/Even addressing model in this chapter.

Color Palette

A **Palette** is like a look-up table. A **Color Palette** is a look-up table for colors. For example, we can store a list of the actual color information in a table. We can then use another table of **indices** into that table:

```
            Index Table | Color Palette
          ------------------------------------
```

```
                0       | red(0), green(0), blue(0)
                1       | red(0), green(0), blue(1)
                2       | red(0), green(1), blue(0)
            ...
```

In the above example, we can reference whatever color we want by just using the index. That saves storage space greatly because after the look up table (The color palette) is created, any time that we want to refer to a color we just use the index.

For example, in a video mode that uses a color palette, video memory acts as the index buffer. So, to draw a pixel using the palette that we created above, just write the **index** of the color that you would like to use:

```
unsigned char* p = 0xa0000;
            p[0] = 0; // black pixel
            p[1] = 1; // blue pixel
            p[2] = 2; // green pixel
```

In the VGA, the Color Palette is handled by the hardware. We can control and change the colors in the palette however way we want. However, because working with the palette requires VGA hardware programming, we will not cover it too much here. Dont worry, we will cover it when we get into VGA hardware.

Palette Animation

Okay, lets take a step back for a moment. Look at the above example again. Notice that the video display will determin what the color is of a pixel by an index. What if, lets say, that the index 1 in the color palette (Like in the above example) changes to a different color? Looking at the above example, index 1 in the color palette is a bright blue color. So, if we are in a palette video mode, any time we write a "1" to someplace in video memory, it will be that bright blue color. This means a simple **memset (vidmem, 1, VIDMEM_SIZE)** will effectively clear video display to this color. Cool, huh?

Knowing that the video display determins what color to display for an index is inside of the Color Palette table, we can change what the color is for any palette entry. This allows us to change the colors on screen by just updating the colors in the palette in some way. This is known as **Palette Animation**.

Palette Animation can create alot of really nice looking and cool effects, such as fire animation, icy effects, etc.

Mode 0x13

Abstract

Video Mode 0x13 is a standard IBM VGA BIOS mode number for a 256 color 320x200 resolution. It uses a 256 **color palette**, did not have square **pixels**, and allowed access to the **Video Memory** as a **Packed-Pixel Framebuffer**. What this means is that it allowed access to video memory as if it was a linear buffer: Just get a pointer to video memory. pointer[0] = pixel 1, pointer[1] = pixel 2, and so on, assuming *pointer* is an *unsigned char**. This is made possible by specific hardware register settings (The video mode "configuation") - Standard VGA does not, by itself, provide access to video memory like this.

The important thing here is that video modes define the resolution, how video memory is accessed, and hardware configuation setup for the operation of that mode. Do not worry if you do not understand everything here; we will go into detail when we cover the VGA hardware in a later chapter.

Because video Mode 0x13 is easy to work with (and fast) I decided to use it for the duration of this chapter. Some other modes require experience with the VGA hardware which I am wanting to avoid in this chapter do to its complexity. Dont worry though, I plan on covering some (Like Mode 12h, 640x480x4 color) later on.

Video Mode 0x13 was used alot in the DOS era for video games do to its simplicity to program and speed. It is a video configuation for a 320 width, 200 height pixel resolution with a 256 color palette. It is a planer video memory mode but acts as a **Linear Frame Buffer (LFB)** which makes it easy to program.

Color Palette

Mode 0x13 has a color palette of 256 colors. Video memory in Mode 0x13 only stores the palette index; the video device will determin what color to render from the installed palette color table. By default, the color table is this:

Mode 13h Color Palette

Here is an example, lookint at the above we can see the first color (0) is black, color 1 is blue, color 2 is green, etc. We can write these colors to video display by using these indices in the above lookup table:

```
unsigned char* p = 0xa0000;

            *p = 0; //black pixel
            *(p++) = 1; // blue pixel
            *(p++) = 4; // red pixel
            *(p++) = 255; //white pixel
```

Compare the above code to the table above and notice how the indices match with the colors in the palette.

Changing the palette

It is possible to change the palette to whatever colors that you would like. However there is not any easy BIOS interrupt for it (Not without using VBE anyways.) Most of the interrupt calls are used to set or get individual or all palette registers which are inside of the VGA **Digital to Analog Converter (DAC)**. This requires some knowledge of the VGA hardware which I am wanting to avoid this chapter for simplicity (Dont worry, I am planning on covering that soon!) Because of this, I decided to wait on covering palette changing (And mabye palette animations) in a future chapter.

The Video BIOS Interface

The VGA Video BIOS Interface is a set of video interrupts (Software interrupt 0x10). Because these are BIOS interrupts, they can only be used in real or v86 modes.

Setting the video mode

INT 0x10 Function 0

You can set the video mode by calling BIOS interrupt 0x10 function 0:

- Input
  - AH = 0
  - AL = video mode
- Output
  - AL = video mode flag (Phoenix, AMI BIOS)
  - AL = CRT Controller (CRTC) mode byte (Phoenix 386 BIOS v1.10)

You will see the CRTC alot more in the future as it is one of the controllers that you will need to program if you plan to directly program the video hardware.

This interrupt can set any text or video mode. For example, the following switches to 320x200x8 bit [mode 0x13]: (**Please remember that all code samples can be found in the demo.**

```
        mode13h:
                mov ah, 0
                mov al, 0x13
                int 0x10
                ret
```

Easy, huh?

The above is all that is needed to get you in a graphics mode. Sure, it will only work in real or v86 mode, but it is as easy as you can get. Dont worry if you do not understand video modes yet; we will cover them a little later.

Getting the video mode

INT 0x10 Function 0xF

You can get the video mode by calling BIOS interrupt 0x10 function 0xF:

- Input
  - AH = 0xF
- Output
  - AH = number of character columns
  - AL = display mode number
  - BH = active page

This interrupt is an easy one and can be used to obtain the current video or text mode. Dont worry about the "active page" part yet. Dont worry if you do not understand video modes yet; we will cover them a little later.

```
getMode:
        mov ah, 0xf
        int 0x10
        ret
```

Other Video BIOS Interrupts

INT 0x10 Function 0xB/BH=1

You can set the palette by calling Video BIOS INT 0x10 function 0xB:

- Input
  - AH = 0xB
  - BH = 1
  - BL = Palette ID
    - 00h background, green, red, and brown/yellow
    - 01h background, cyan, magenta, and white

This interrupt may not be supported on all systems.

INT 0x10 Function 0xC

You can write a pixel to the display using this interrupt.

- Input
  - AH = 0xC
  - BH = Page number
  - AL = Pixel color
    - if bit 7 set, value is XOR'ed onto screen except in 256-color modes
  - CX = column
  - DX = row
- Output
  - AL = pixel color

This interrupt can only be used in graphics modes.

INT 0x10 Function 0xD

You can read the pixel by calling Video BIOS INT 0x10 function 0xB:

- Input
  - AH = 0xC
  - BH = Page number
  - CX = column
  - DX = row

This interrupt will only work on graphics modes.

Primitives

Plotting your first pixel

"The secret to making any video game is the ability to change the color of a pixel." - Teej

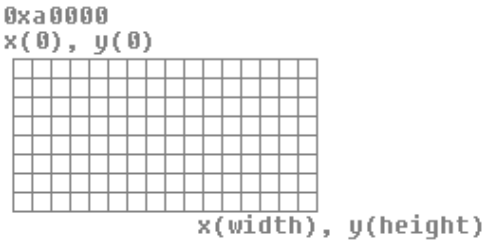
We have covered alot in this chapter and have yet to draw a pixel on screen. Whats up with that? I decided to close this chapter with the basics of most basic graphics primitives - rendering a pixel to the screen.

Because we are working in Mode 0x13, remember that it acts like a linear frame buffer. So vidmem[0] is the first byte of video memory, vidmem[1] is the second byte. Also, remember that Mode 0x13 uses a byte for each pixel as an index into the Color Palette. This means that, we can write a pixel easy like this:

```
unsigned char* p = 0xa0000;
            p[0] = 1; // blue pixel
```

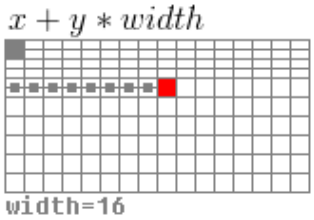
Cool, huh? Thats all that is needed and you have a pixel!

It is easier to think in terms of the **cartesian coordinate system**. In this system, we use coordinates, such as X and Y to represent its location on a 2d graph like this:



The top-left corner of video memory is at **v = [0,0]** where **v** is a 2d vector. This is the first pixel in video display. The last byte is at **v = [width, height]**. Assuming each coordinate is a pixel, we can come up with a formula that allows us to be able to draw a pixel at any location on screen.

Lets say we start at **v = [0,0]** in the graph above. If we add **width** to our position, we always end up right below where we were. For example, in the above graph, **width = 16**. Assuming we started at the top-left corner, counting 16 to the right, you will find yourself right below (on the next line) from where you started. Because of this, we can calculate **y** by doing **y * width**. Afterwords we can just add **x** (The offset in that line) and we have our formula:



To render a pixel at any **[x,y]** location, we use the formula **x + y * width**. With this, we can create a simple routine like this:

```
;-----------------------;
;   renders pixel
;   cl = color ax = y bx = x
;   es:bp = buffer
;-----------------------;
pixel:
; [x + y * width] = col

        pusha
        mov di, VGA_MODE13_WIDTH
        mul di ; ax = y * width
        add ax, bx ; add x
        mov di, ax
        mov byte [es:bp + di], cl ; plot pixel
        popa
        ret
```

es:bp points to the video display, or another buffer that we want to render to. cl is the color index that you want to use, ax is the Y location and bx is the X location.

Clearing the screen

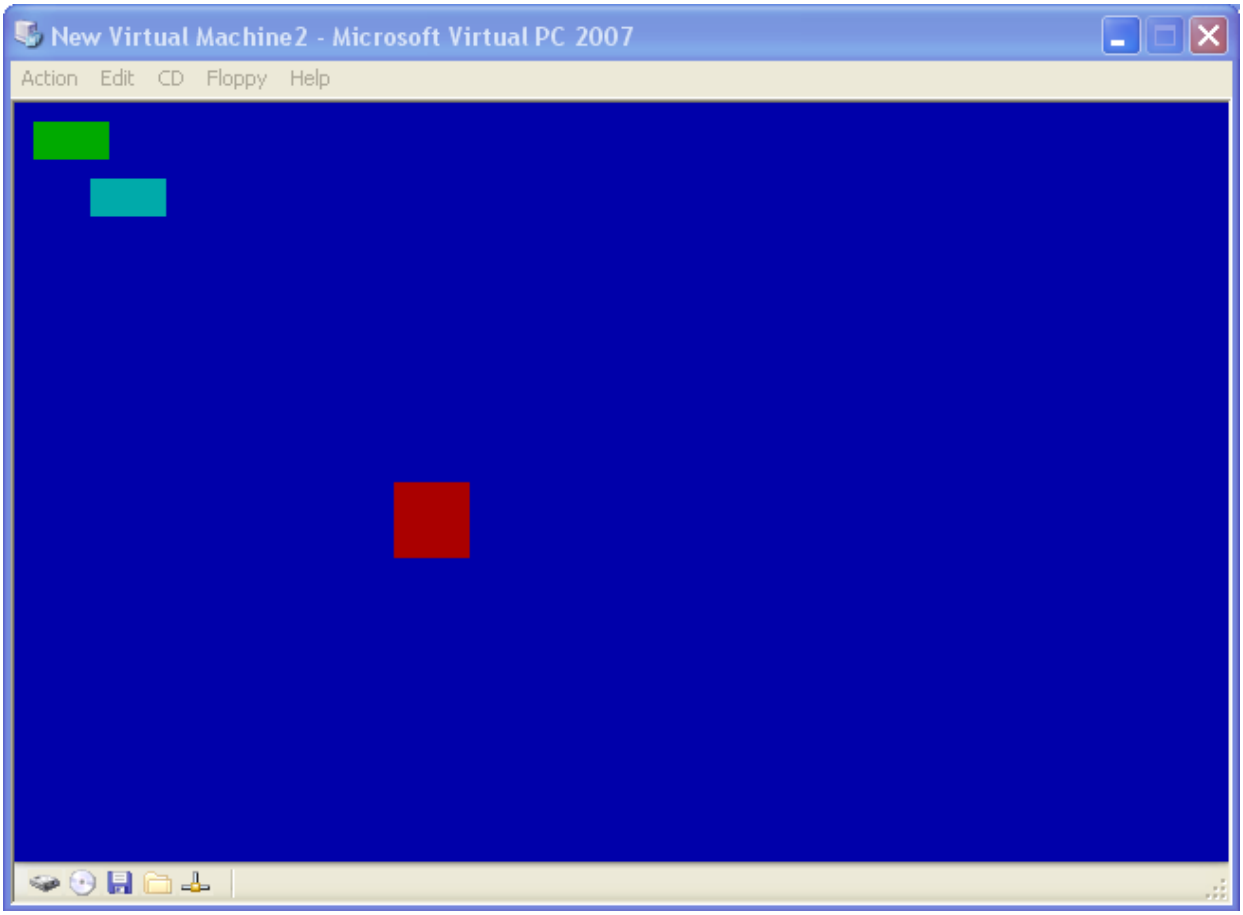
There are several ways of clearing the screen. This is important as alot of times when switching video modes, you may see alot of garbage on screen.

One method that we can do is just call our pixel routine above **width * height** times. A better method would be to write multiple pixels at once. Knowing, for example, that the size of a pixel in Mode 13h is a byte, we can easily store 2 bytes (2 pixels) in a word size register and use that instead:

```
;----------------------------;
;   clear screen
;   cl = color
;----------------------------;
clrscr:

        pusha
                mov dl, cl  ; dx = 2 pixels
                mov dh, cl

                mov cx, 0
                xor di, di

        .l:
                mov word [es:bp + di], dx ; plot 2 pixels
                inc di ; go forward 2 bytes
                inc di
                inc cx
                cmp cx, (VGA_MODE13_WIDTH * VGA_MODE13_HEIGHT) / 2 ;end of display?
                jl .l

        popa
        ret
```

es:bp refers to either video memory or another buffer and cl is the color that you would like to use.

Demo



Mode 13h Demo running in real mode

Demo Download

This demo spices what we talked about a little by adding an additional routines: **line**, which renders a horizontal line, and is used to render the rectangles in the demo.

Conclusion

That's all for this chapter!

The next chapter will cover VESA VBE and how we can use it to work with high resolution graphics modes. We will also cover the Super VGA, Bank Switching, and a few more graphics concepts including **Double and Triple Buffering,** and **Page Flipping**. Thats right, we are going high resolution with VBE :)

We will also be going back to C in the next chapter and cover some more graphics primitives. I still plan on covering VGA hardware however that will be after VBE. VGA hardware is quite overcomplicated; I want to hold off on the more complex topics in graphics and VGA until a little later.

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System

Questions or comments? Feel free to Contact me.

Would you like to contribute and help improve the articles? If so, please let me know!

Home

BrokenThorn Entertainment

Operating System Development Series

This series is intended to demonstrate and teach operating system development from the ground up.

Graphics 2: VGA and SuperVGA

"Controlling complexity is the essence of computer programming" - Brian W. Kernighan

by Mike, 2013

1. Introduction

Welcome!

In the previous article, we introduced some concepts pertaining to video devices with an emphasize on VGA on the Video BIOS firmware. In this article, we will look at different hardware and firmware interfaces for video devices for VGA, SuperVGA and Video BIOS support and implement a common video interface in C supporting the interfaces. Here are the topics to be covered.

1. VGA hardware
2. VGA BIOS
3. Vesa BIOS Extensions
4. Bochs VBE Interface

We first introduce the **Video Graphics Array (VGA)** standard as it is the most supported on personal computers and one of the oldest. The VGA standard provides a way to interact with video hardware in a standard way, but it is limited to low resolution display modes and lack of graphics acceleration support present in modern display devices. We will look at both the hardware interface and the VGA BIOS firmware interface. The VGA BIOS interface is by far simpler then the VGA hardware interface but can only be used in real or v86 processor modes. The hardware interface can be used from any processor mode.

Then we move on to **Vesa BIOS Extensions (VBE)**. VBE is a standard developed by the **Video Electronic Standards Association (Vesa)** that provides a standard set of BIOS extensions for supporting high resolution display modes and monitor features. Due to it being a BIOS extension, not all personal computers support it. It also can only be used from real or v86 modes.

In summary, the goal of this chapter is to cover changing the video mode and accessing display memory. By the end of the chapter, you should have a demo built that does just that, either using VGA or SuperVGA. Later chapters will then focus on the graphics (and possibly some SuperVGA hardware graphics support.)

1.1. Interfacing with the BIOS

Before we start getting into the main topics of the chapter, we need to take a small detour and take a closer look at the BIOS. You may recall our use of some of these BIOS services for VGA in the previous article. If the software uses any of these services then it must run in real or v8086 mode. This poses a problem for protected mode or long mode software. Thus we need to find a way to resolve this before continuing. **If you don't plan on using any BIOS services, however, please feel free to skip this section.**

There are two approaches of calling the BIOS from protected mode (not long mode.)

1. Drop down to real mode and call the BIOS
2. Virtual 8086 mode

The first method is simpler but can be very ugly for more complete designed systems. The second approach is the most commonly used method but is also the hardest; requiring user mode, interrupt dispatching, task switching, and instruction emulation.

Method 1

The first method requires that the software be able to switch to real mode from protected mode when needed. Without certain restrictions applied to the software design (such as not supporting virtual memory or higher half kernel support) this method can become increasingly complicated to support to the point where it is not worth it. It is, however, the simplest method to implement and requires the least amount of additional software support. For these reasons, we opted to go with this method for the associated demos but highly recommend using method 2 when the software system is large enough to warrant the need.

In order to implement this method, we need a routine or a set of routines to act as an interface between 32 bit protected mode and 16 bit real mode. These routines must do the following while preserving routine input and output values.

1. Save current system state that must be reserved. In the most basic case, this is the protected mode stack and IDTR.
2. Disable hardware interrupts (CLI instruction).
3. Reload original IVT. This is done by setting IDTR.size to 0xffff and IDTR.base to 0.
4. Perform a jump to 16 bit protected mode.
5. Disable protected mode by clearing CR0.PM bit
6. Perform a jump to 16 bit real mode code.
7. Set all real mode segments, enable interrupts and call BIOS.
8. Perform a jump to 32 bit protected mode.

9. Restore saved system state. In the most basic case as in (1) this is the selectors, protected mode stack, and IDTR.

This routines can get very complicated as the system becomes more demanding in what it supports. The above list sounds like a lot, but its more tricky then hard—provided the system does not use paging and the kernel image is less then 1MB. In other words, **we assume the project base address is 64K and paging is disabled in order to keep the routine relatively simple.**

In the demos, the method **io_services** is used to call the BIOS. It drops into real mode using the steps above. **io_services** looks like this:

```
extern void io_services (unsigned int num, INTR* in, INTR* out);
```

Where **num** is the interrupt number, **in** is a pointer to a **INTR** structure, and **out** is a pointer to an output **INTR** structure. **INTR** is a set of structures that store register values. Both **io_services** and **INTR** are fairly large and so will be omitted in the text. Please reference **bios.asm** in the demos.

Example. This example uses the function **io_services** and **INTR** structure to call the BIOS to set the video mode. Note the C code is runs in protected mode.

```
void vga_set_mode (int mode) {
        /* call BIOS */
        INTR in, out;
        in.eax.val = mode;
        io_services (0x10, &in, &out);
}
```

If the software is in long mode, the only options is to program the device directly or to write an emulator.

Method 2

The second method is using v8086 mode. This is by far the best long-term way to support calling the BIOS firmware but is also the most demanding. At a minimum, v8086 mode requires that the operating system support the following.

1. User mode processes
2. Task switching
3. Interrupt dispatching
4. Instruction emulation

Virtual 8086 mode can only execute as user mode processes. This poses a problem however as user mode processes can not execute kernel mode instructions (like **int**) and so can not call the BIOS which sort of defeats the purpose. In other words, when the v8086 process executes an **int** (interrupt) instruction, it triggers a **general protection fault (GPF)**. What can we do to fix this?

We are not entirely out of solutions here. While the v8086 process cannot call the BIOS, the kernel *can*. When the GPF occurs, the kernel effectively gets called. The kernel GPF handler then can detect what caused the GPF and do something about it like so:

1. Check current process.v8086 flag
2. If set; call **v86_monitor**
3. If not set, continue with GPF and possibly terminate process

Our **v86_monitor** is a special function that is called by the kernel GPF handler for all v8086 processes. Now we are getting somewhere; what we have here is that whenever the v8086 process attempts to call the BIOS (or use any kernel mode instruction) the kernel GPF handler gets called which calls **v86_monitor** to "monitor" the v8086 task.

The **v86_monitor** implements a **v8086 monitor** that is responsible for *emulating* the problem instructions the v8086 task tried to use. For example, the **v8086 monitor** would *detect* the problem instruction (it will have the CS:EIP given by the CPU) as an interrupt call and *emulate* it by invoking IVT [n] where n = BIOS number to call (recall however that the IVT instruction pointers are in segment:offset format *not* linear.)

2. Video Graphics Array (VGA)

Readers may skip this section if you only want to look at SuperVGA.

The **Video Graphics Array (VGA)** is the design of display hardware first introduced in 1987 for the IBM PS/2 computers [1] but has been widely adopted by organizations as a display standard. The highest video mode resolution supported is 640x480x16 color. Due to the widespread adoption by PC manufactures, VGA has become one of the oldest standards still supported by modern PC's.

VGA was followed by IBM's **Extended Graphics Array (XGA)** standard but extensions implemented by different manufacturers produced **SuperVGA** adapters that are common in modern PC's. Most SuperVGA cards are backward compatible with the VGA standard.

Please note that this is not going to be an exhaustive explanation of VGA due to its complexity. Please reference resources [2] and [3] for more information on VGA hardware and I encourage reading one of the many large books on VGA.

2.1. Video modes

There is a standard set of video modes and mode numbers supported by VGA. The **video mode** refers to the **display configuration** and its properties, such as **resolution**, **bit depth** (bits per pixel), number of colors, **memory mode**, etc. **Standard video mode numbers are 0h, 1h, 2h, 3h, 4h, 5h, 7h, Dh, Eh, Fh, 10h, 11h, 12h, and 13h.** There is nothing special about the mode number itself; it is just used by the video BIOS to refer to a particular video mode. There may be more modes, however they are nonstandard.

Mode	Resolution	Color depth	Mode	Resolution	Color depth

0h	40x25 Text	16 Color	Dh	320x200	16 Color
1h	40x25 Text	16 Color	Eh	640x200	16 Color
2h	80x25 Text	16 Color	Fh	640x350	2 Color
3h	80x25 Text	16 Color	10h	640x350	16 Color
4h	320x200	4 Color	11h	640x480	2 Color
5h	320x200	4 Gray	12h	640x480	16 Color
7h	80x25 Text	2 Color	13h	320x200	256 Color

The highest resolution supported by standard VGA is **mode 12h** which is 640x480x16 color. (Interesting note, Windows XP logo screen runs in mode 12h.) Higher resolutions can only be obtained using SuperVGA which is described later on.

2.2. VGA Firmware

We first look at the firmware interface of VGA, and the facilities provided by the **Video BIOS**. In the last article we introduced some of the facilities, most notably **interrupt 0x10 function 0** to set the video mode. The Video BIOS provides services for setting, getting, and working with VGA hardware through a more abstract interface. Arguably it is far more safer, simpler, and more portable for software to use the video BIOS facilities then it is to directly control the hardware.

Here we present some of the common facilities that software can use for video services. Of course, because these are BIOS interrupts, they can only be used in real or v8086 mode and on systems that have BIOS firmware. Also, they can only be used by kernel mode software.

INT 0x10 Function 0 – Set Video Mode

Input:

- AH=0
- AL = video mode

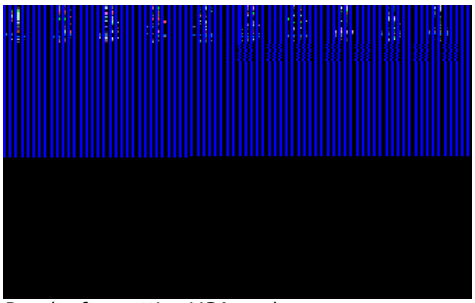
Output:

- AL=video mode flag (Phoenix, AMI BIOS)
- AL = CRT Controller (CRTC) mode byte (Phoenix 386 BIOS v1.10)

Example. This function sets the video mode.

```
void vga_set_mode (int mode) {
        /* call BIOS */
        INTR in, out;
        in.eax.val = mode;
        io_services (0x10, &in, &out);
}
```

We can now use the above function to set any VGA BIOS video mode, such as Mode 13h. The result might look something like this.



Result after setting VGA mode.

As you can see from the above image, there is a lot of garbage on the display. This "garbage" is actually what what was in VGA memory before the mode switch. This includes everything from all 4 **planes** (we will look at these later on) such as any textural characters from plane 1 and the VGA font in plane 2. If we clear the display, all of this is cleared. This is a problem if the software must return back to text mode because by clearing the display the software cleared the VGA font. There are two ways to fix this:

1. Upload a new copy of the VGA font from the BIOS
2. Save the VGA font before clearing memory and restoring it later.

By performing one of the above the software can switch to a graphics mode and return back to text mode without error.

INT 0x10 Function B – Set Palette (Text modes only)

Input:

- AH=B
- BH=1

- BL=Palette ID.
  - 00h background, green, red, and brown/yellow
  - 01h background, cyan, magenta, and white

Example. The following code sets a VGA palette.

```
void vga_set_palette (int id) {
        /* call BIOS */
        INTR in, out;
        in.eax.val = 0xB;
        in.ebx.val = 0x0100 | id;
        io_services (0x10, &in, &out);
}
```

INT 0x10 Function C – Write pixel

Input:

- AH=C
- BH=Page number
- AL=Pixel color
- CX=Column
- DX=Row

Output:

- AL=Pixel color

Example. The following writes a pixel. This might be omitted in the demos as it shouldn't be used; we only provide it for completeness.

```
void vga_plot_pixel (int col, int x, int y) {
        /* call BIOS */
        INTR in, out;
        in.eax.val = 0x0C00 | col;
        in.ebx.val = 0;
        in.ecx.val = x;
        in.edx.val = y;
        io_services (0x10, &in, &out);
}
```

Note that some versions of the Bochs emulator do not support this interrupt. It is however implemented for VirtualPC. We can use the above interrupt to plot pixels.

INT 0x10 Function F – Get Video Mode

Input:

- AH=F

Output:

- AH=Number of character columns
- AL=Display page number
- BH=Active page

Example. The following code returns the mode information.

```
void vga_get_mode (unsigned int* col, unsigned int* dispPage, unsigned int* actPage) {
  INTR in, out;
  /* sanity check */
  if (!col || !dispPage || !actPage)
     return;

  in.eax.val = 0xf;
  io_services (0x10, &in, &out);
  *dispPage = out.ax.r.al;
  *actPage = out.bx.r.bh;
}
```

2.3. VGA Hardware Interface

The VGA hardware interface is very complex, consisting of five controllers and over a hundred hardware registers accessible from the Port I/O address space. A standard set of register configurations help define the standard VGA video modes we still see today. Due to

the use of the Port I/O address space, software can interface with the VGA hardware in any processor mode. Of course, the software must be in supervisor level (ring 0) for the **in** and **out** family of instructions to actually interact with the hardware. Thus, only kernel mode software can access VGA hardware. User mode software that attempts it will generate a general protection fault when executing the **in** or **out** instruction.

Before looking at the hardware, there is a warning. The warning applies to all display monitors and video cards that may lack protection from invalid data or data outside of the range supported by the device (for example, an excess frequency setting that the device is unable to work with in a safe manner.) Most modern monitors will display an error message or show no output on invalid settings. Make sure to test the driver software in an emulator and virtual environment first and verify the driver software works properly before testing on real hardware.

Video memory

The memory layout of VGA differs depending on the type of video mode is currently active. The VGA supports the following memory layouts.

1. Linear
2. Planar
3. Palette
4. 4 Color mode

We will take a look at all of these modes next and when they are used. Knowing the different memory models is important in order to properly read and write to display memory. We will cover linear modes last as it is the most complex. Before that though, we first need to know how to access video memory.

Recall the **memory map** that we have looked at in the **system architecture** chapter. Here it is again.

- **0x00000000 - 0x000003FF** - Real Mode Interrupt Vector Table
- **0x00000400 - 0x000004FF** - BIOS Data Area
- **0x00000500 - 0x00007BFF** - Unused
- **0x00007C00 - 0x00007DFF** - Our Bootloader
- **0x00007E00 - 0x0009FFFF** - Unused
- **0x000A0000 - 0x000BFFFF** - Video RAM (VRAM) Memory
- **0x000B0000 - 0x000B7777** - Monochrome Video Memory
- **0x000B8000 - 0x000BFFFF** - Color Video Memory
- **0x000C0000 - 0x000C7FFF** - Video ROM BIOS
- **0x000C8000 - 0x000EFFFF** - BIOS Shadow Area
- **0x000F0000 - 0x000FFFFF** - System BIOS

According to the memory map, video memory is mapped into the physical address space at **0xA0000 – 0xBFFFF**. This is **0x1FFFF** bytes of memory or 131072 bytes which is 128K. According to [2] VGA hardware has up to 256K or memory however only 128K of it is mapped. The unmapped memory is accessible by changing the address decoding mechanism of the VGA (we will not cover that here though.) **Graphics mode memory starts at 0xA0000 and may extend to 0xAFFFF, or 64K.** Some of the following examples will demonstrate this.

Planar memory modes (16 bit color modes)

VGA memory is referenced as 4 **planes** each 64k of memory. You can think of them as different memory banks that are connected to each other (although that might not be the case.) Its sort of like having 4 windows on top of each other. Those windows are the planes.

In 16 color modes there are 4 bits per color. The 4 bits are stored at the same location on each plane.

Example.	A 4 bit pixel can be stored at plane0[0], plane1[0], plane2[0] and plane3[0]. This displays a single pixel to display memory. Notice that the 4 bit pixel is stored across all 4 planes, where each plane only stores one bit of the pixel. Also notice that the pixel is at the same location (index is 0) of all 4 planes.

We will show an example of how to write a pixel in planar memory modes a little later after reviewing the registers and how to set a video mode. Unfortunately writing a pixel requires also writing to a hardware register to select the plane to write to and thus an example cannot be provided yet.

Palette memory modes (256 color modes)

In palette memory modes, each pixel is represented by a number—an *index*—into a color table. The color table is the **palette**. Probably most notable is the 256 color mode, where each pixel is 8 bits. Other palette modes include the 16 color (4 bit pixel) and monochrome (1 bit pixel) modes that have only 2 colors.

The color table might look something like this. This is the palette used in a 16 color mode; in fact, its the VGA text mode palette that is the default on system start up.

Index	Color name	Index	Color name
0	Black	8	Dark Gray
1	Blue	9	Light Blue

2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	Light Gray	15	White

Lets look at an example of the palette use in text modes.

Example. Recall that in text modes, each character includes a character code (ASCII character typically) and an attribute byte. The attribute byte is an index into the above palette. In other words, if you have worked with text modes in protected mode you have already been working with a palette memory mode!

256 color modes were widely used by DOS video games. This became infamously known as **Mode 13h** after its standard Video BIOS mode number. Mode 13h is an interesting mode because it is very fast, linear, and can display 256 colors on the screen at once. In other words, video memory in this mode is **linear** where each pixel is 1 byte and stored right after each other in memory. What makes Mode 13h interesting is that VGA is a planar display device not a linear one.

Example. The following C code defines a function, **pixel_256** that plots a pixel in 256 color mode at some x and y location in mode 13h. Recall that mode 13h is 320x200 and so the pitch (here pitch is width) is 320. Also unlike the VGA BIOS interrupt plot pixel service, this one will work in any PC emulator or virtual machine.

```
#define VGA_VRAM 0xA0000
#define PITCH 320
void pixel_256 (unsigned char color, unsigned int x, unsigned int y) {
    unsigned char* fb       = (unsigned char*) VGA_VRAM;
    unsigned int offset     = y * PITCH + x;
    fb [offset] = color;
}
```

The above example also demonstrates the simplicity of working with Mode 13h. We can render multiple pixels by just writing to them consecutively. In 256 color modes each pixel is represented as a byte so we use unsigned char's.

In palette memory modes, there is a maximum to the amount of colors that can be displayed at once. This is the same as the number of entries in the color table being used for the mode. For example, 256 color mode can only display at most 256 colors on the screen at once. 16 color modes can only display 16 colors. Software could modify the color table itself in order to display different colors.

Linear memory modes

Linear memory modes are the set of video modes that support a **linear frame buffer (LFB)**. Linear memory is an array of consecutive bytes; like an array in C. Mode 13h is an example mode that has a linear memory model. Another example is the infamous **Mode X (360x480x256 color)** and **Mode Q (chain-4 256x256x256 color)** which are modified versions of Mode 13h.

Recall that VGA is not a linear device; it does not support linear memory modes. Modes like Mode 13h, Mode X, and Mode Q are all planar modes that configure the hardware in a way that creates the illusion of a linear memory model.

To write a pixel in linear memory modes, all we need to do is calculate the offset of the pixel location and write the pixel to the frame buffer.

Example. The following code plots a 8 bit pixel in a linear memory mode. This example is the same as the above but is more generic; it can be applied to any mode that has a linear memory model.

```
#define VGA_VRAM 0xA0000
#define PITCH 320
void pixel_256 (unsigned char color, unsigned int x, unsigned int y) {
    unsigned char* fb       = (unsigned char*) VGA_VRAM;
    unsigned int offset     = y * PITCH + x;
    fb [offset] = color;
}
```

Hardware

The VGA hardware design consists of the following components. Some of these might look familiar.

1. CRT Controller
2. Sequencer
3. Graphics Controller
4. RAMDAC
5. Video memory
6. Attribute Controller

All of the hardware registers are mapped into the I/O port space. That is, we can access them using the **in** and **out** family of CPU instructions. Most of the controllers map two registers: an **address register** and **data register**. The **address register** stores an **index** of a particular register that we want to read or write from, the **data register** contains the data of that register. This may become more clear as we see some examples.

Before looking at each of these components, we would like to emphasize the complexity of VGA hardware. Entire books have been written that covers VGA in depth; covering all of the details in a single article is a very daunting task. In order to keep the discussion minimal, we will only look at what each component does and the set of registers used by the component. Due to the large number of registers, we will not be describing the registers here. Unfortunately we cannot describe how to set a video mode without having some background in the hardware registers. To rectify this, we present a list of all of the registers and a table of the video mode settings will

be listed at the end. **This will allow readers to set VGA video modes without having to worry about the details of each register nor its format**. We do encourage reading more about each register by referencing [3] and [4] however for anyone that is interested in learning more about VGA.

In short, don't worry to much on the details of the registers. The video mode list will tell you what value to put into what register.

CRT Controller (CRTC)

Warning.	**Improperly configuring the CRT Controller (CRTC) can potentially damage the video card or attached monitor**. Although rare, instances of CRT and LCD monitors have been known to burn out or explode due to improper or erroneous configuration.

The **CRT Controller (CRTC)** is responsible for controlling the output of video data to the display monitor. It is accessed by an address and data register. **Address register is at 3D4h the Data Register is at 3D5h (or 3B4h, 3B5h respectively if Miscellaneous Output Register I/O Address select field is set.)** We can select what CRTC register we want to access by using the address register. We can then use the data register to read or write from that CRTC register.

We will not be providing a complete technical review of each register due to the number of registers. Please reference the FreeVGA project page dedicated to describing the CRT registers in detail. If there is demand, we may extend our topic on VGA in the future.

0	Horizontal total register
1	End Horizontal display register
2	Start horizontal blanking register
3	End horizontal blanking register
4	Start horizontal retrace register
5	End horizontal retrace register
6	Vertical total register
7	Overflow register
8	Preset row scan line register
9	Maximum scan line register
10	Cursor start register
11	Cursor end register
12	Start address high register
13	Start address low register
14	Cursor location high register
15	Cursor location low register
16	Vertical retrace start register
17	Vertical retrace end register
18	Vertical display end register
19	Offset register
20	Underline location register
21	Start vertical blanking register
22	End vertical blanking register
23	CRTC mode control register
24	Line compare register

The CRTC registers allow us to control the pixel clock timings of the horizontal and vertical retrace and blanking periods of the display. These timing periods help control the output of the display (the **video resolution**), and the **refresh rate**. Other CRTC registers allow us to change the start address of video memory (the **preset row scan line** and **start address** registers, hardware cursor location (**cursor location** registers) and underline (**underline location** register.) The **CRTC mode control** register gives us some way of controlling the CRT itself and some addressing modes.

Graphics Controller

The graphics controller is responsible for managing the interface between CPU and video memory. **The address register is mapped to 3CEh and the data register is mapped to 3CFh**. In order to access the graphics controller registers, write the index of the register into the address register and read or write from the data register.

The following is a list of the standard registers. Please reference the FreeVGA project for a detailed description of each register.

0	Set/Reset register
1	Enable Set/Reset register
2	Color compare register
3	Data rotate register
4	Read map select register
5	Graphics mode register
6	Miscellaneous graphics register
7	Color don't care register
8	Bit mask register

Sequencer

The sequencer manages the interface between the video data and RAMDAC. **The address register is mapped to 3C4h and the data register is mapped to 3C5h**. In order to access the sequencer registers, write the index of the register into the address register and read or write from the data register.

The following is a list of the standard registers. Please reference the FreeVGA project for a detailed description of each register.

0	Reset register
1	Clocking mode register
2	Map mask register
3	Character map register
4	Sequencer memory mode register

Attribute Controller

The attribute controller consists of 21 registers for the VGA. The controller has two registers mapped into I/O port space, one at **3C0h** and the other at **3C1h**. Unlike the other controllers, **the attribute controller uses the register at 3C0h as both a data write port and address port**. **The register at 3C1h is the data read register**. In order to communicate with the attribute controller, **the software should first write a register address to port 3C0h followed by the data to write to that register**. **Data can be read by writing to port 3C0h and reading from 3C1h after**. The attribute address register also has a particular format.

Attribute Address Register							
7	6	5	4	3	2	1	0
		PAS	Address				

Address refers to a register index to access. Please refer to the following table of registers.

PAS (Palette Address Source) Determines access to palette data by host or EGA display adapter. If it is 0, the host can access the palette RAM and the adapter will disallow display memory from gaining access to the palette. If 1, the display memory can access the palette RAM and the host is disallowed from accessing it.

The attribute controller stores a set of 16 palette registers that map a color index to a color. This small palette is a drawback from EGA. VGA still uses these registers but instead of storing the palette information it stores an address into a second set of color index registers. We won't be covering the details of how it all works – but I encourage reading [4] page 394 for anyone that is interested.

The following is a list of the standard registers. Please reference the FreeVGA project or [4] for a detailed description of each register.

0-15	Palette entry register
16	Mode control register
17	Overscan color register
18	Color plane enable register
19	Horizontal pixel panning register
20	Color select register

General Registers / External Registers

These register lists just don't seem to end do they? We have looked at lists of registers for all of the major controllers of VGA however there are more sets of registers for general use and miscellaneous data that are used by the video modes and thus need to be looked at. These registers can be referred to as **general registers** or **external registers**.

The following is a list of the standard registers. Please reference the FreeVGA project or [4] for a detailed description of each register. **Please note that we assume a color adapter and VGA (not EGA).** Monochrome and EGA adapters may use different ports.

Write port: 3C2h Read port: 3CCh	Miscellaneous output register
Write port: 3BAh Read port: 3CAh	Feature control register
Read port: 3C2h	Input Status #0 register
Read port: 3BAh	Input Status #1 register

Color Registers

These registers allow us the software to manipulate and manage the 256 color palette. They are an important set of registers if working in 256 color modes and may be used after setting the video mode so getting familiar with them is a good idea. You might also be happy to learn that this is the final set of registers that we are going to look at.

The following is a list of the standard registers. Please reference the FreeVGA project or [4] for a detailed description of each register.

Write port: 3C8h Read port: 3C8h	PEL address write mode register
Write port: 3C7h	PEL address read mode register
Write port: 3C9h Read port: 3C9h	PEL data register
Read port: 3C7h	DAC state register
Write port: 3C6h Read port: 3C6h	PEL mask register

2.4. Standard video modes

By configuring the video hardware we can define different display modes with different properties. However many modes that the video hardware can support many monitors do not. Other configurations may have undesired effects. Some VGA mode configurations have become a standard; well supported by most monitors. The following is a list of standard video modes and their configurations. **Standard video mode numbers are Mode 0h, 1h, 2h, 3h, 4h, 5h, 7h, Dh, Eh, Fh, 10h, 11h, 12h, and 13h**. Other notable modes, such as **Mode X** and **Mode Q** while well supported are not standard modes.

Please reference [3] or [4] for other lists. The following tables were adopted from pages 304-305 of [4]. The tables are split between the different sets of registers each mode modifies. The top row indicates the mode number, the left column is the index to use to access the particular register. According to [4] the values presented represent the **initial mode state** the standard video BIOS uses.

In order to set a video mode, the software must disable the video output and write the associated values to all of the hardware registers that it modifies. This sets everything – the read/write modes, address start, horizontal and vertical blanking periods and timing, refresh rate, resolution, graphics mode type, and more.

While every attempt has been made to insure accuracy, there might be some errors that have been undetected in the following tables. The tables were written to match that of which what was presented in [4] but may have errors in the rewrite. We give all credit to the original authors.

General registers

	Mode														
Index	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	63	63	63	63	63	63	63	A6	63	63	A2	A3	E3	E3	63
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	70	70	70	70	70	70	70	70	70	70	70	70	70	70	70
3	4	4	5	5	4	4	5	FF	4	4	FF	4	4	4	4

Sequence registers

	Mode														
Index	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
1	9	9	1	1	9	9	1	0	9	1	1	1	1	1	1
2	3	3	3	3	3	3	1	3	F	F	F	F	F	F	F
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	2	2	2	2	2	2	6	2	6	6	6	6	6	6	E

CRTC Registers

	Mode														
Index	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	2D	2D	5F	5F	2D	2D	5F	FF	2D	5F	FF	5F	5F	5F	5F
1	27	27	4F	4F	27	27	4F	FF	27	4F	FF	4F	4F	4F	4F
2	28	28	50	50	28	28	50	FF	28	50	FF	50	50	50	50
3	90	90	82	82	90	90	82	FF	90	82	FF	82	82	82	82
4	2B	2B	55	55	2B	2B	54	FF	2B	54	FF	54	54	54	24
5	A0	A0	81	81	80	80	80	FF	80	80	FF	80	80	80	80
6	BF	BF	BF	BF	BF	BF	BF	FF	BF	BF	FF	BF	B	B	BF
7	1F	1F	1F	1F	1F	1F	1F	FF	1F	1F	FF	1F	3E	3E	1F
8	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0
9	C7	C7	C7	C7	C1	C1	C1	FF	C0	C0	FF	40	40	40	41
A	6	6	6	6	0	0	0	FF	0	0	FF	0	0	0	0
B	7	7	7	7	0	0	0	FF	0	0	FF	0	0	0	0
C	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0
D	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0
E	0	0	0	0	0	0	0	FF	0	0	FF	0	0	0	0
F	31	31	59	59	31	31	59	FF	31	59	FF	59	59	59	31
10	9C	9C	9C	9C	9C	9C	9C	FF	9C	9C	FF	83	EA	EA	9C
11	8E	8E	8E	8E	8E	8E	8E	FF	8E	8E	FF	85	8C	8C	8E
12	8F	8F	8F	8F	8F	8F	8F	FF	8F	8F	FF	5D	DF	DF	8F
13	14	14	28	28	14	14	28	FF	14	28	FF	28	28	28	28
14	1F	1F	1F	1F	0	0	0	FF	0	0	FF	F	0	0	40
15	96	96	96	96	96	96	96	FF	96	96	FF	63	E7	E7	96
16	B9	B9	B9	B9	B9	B9	B9	FF	B9	B9	FF	BA	4	4	B9
17	A3	A3	A3	A3	A2	A2	C2	FF	E3	E3	FF	E3	C3	E3	A3
18	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

Graphics Controller

	Mode														
Index	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	10	10	10	10	30	30	0	10	10	0	0	10	0	0	40
6	0E	0E	0E	0E	0F	0F	0D	0A	5	5	5	5	5	5	5
7	0	0	0	0	0	0	0	0	0	F	5	0	5	F	F
8	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF	FF

Attribute Controller

	Mode														
Index	0	1	2	3	4	5	6	7	D	E	F	10	11	12	13
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	13	13	17	8	1	1	8	1	3F	1	1
2	2	2	2	2	15	15	17	8	2	2	0	2	3F	2	2
3	3	3	3	3	17	17	17	8	3	3	0	3	3F	3	3
4	4	4	4	4	2	2	17	8	4	4	18	4	3F	4	4
5	5	5	5	5	4	4	17	8	5	5	18	5	3F	5	5
6	6	6	6	6	6	6	17	8	6	6	0	14	3F	14	6
7	7	7	7	7	7	7	17	8	7	7	0	7	3F	7	7
8	10	10	10	10	10	10	17	10	10	10	0	38	3F	38	8
9	11	11	11	11	11	11	17	18	11	11	8	39	3F	39	9
A	12	12	12	12	12	12	17	18	12	12	0	3A	3F	3A	0A
B	13	13	13	13	13	13	17	18	13	13	0	3B	3F	3B	0B
C	14	14	14	14	14	14	17	18	14	14	0	3C	3F	3C	0C
D	15	15	15	15	15	15	17	18	15	15	18	3D	3F	3D	0D
E	16	16	16	16	16	16	17	18	16	16	0	3E	3F	3E	0E
F	17	17	17	17	17	17	17	18	17	17	0	3F	3F	3F	0F
10	8	8	8	8	1	1	1	0E	1	1	0B	1	1	1	41
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0F	0F	0F	0F	3	3	1	0F	0F	0F	5	0F	0F	0F	0F
13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

To set a video mode, we must set all of the registers the mode modifies in the above tables. The following example illustrates setting the mode.

Example. In order to set a video mode, we need to upload all of the values into the registers associated with that mode. For example, if we have a function, **uploadRegisters**, that does this, we can set Mode 13h by doing the following.

```
unsigned char _mode13h = {
   /* general registers */
   0x63,0,0x70,0x4,
   /* sequencer */
   0x3,0x1,0xF,0,0xE,
   /* CRTC */
   0x5F,0x4F,0x50,0x82,0x24,0x80,0xBF,0x1F,0,0x41,
   0,0,0,0,0,0x31,0x9C,0x8E,0x8F,0x28,0x40,0x96,0x89,0xA3,0xFF,
   /* graphics */
   0,0,0,0,0,0x40,0x5,0xF,0xFF,
   /* attribute */
   0,1,2,3,4,5,6,7,8,9,0xA,0xB,0xC,0xD,0xE,0xF,0x41,0,0xF,0,0
}

/* set mode 13h */
uploadRegisters (&_mode13);
```

After writing the register values, we have successfully changed video modes.

Demo running in VGA Mode 13h and clearing display memory

Recall that we have looked at the planar memory model earlier; we could not complete an example of plotting a pixel until looking at the hardware registers. We now present that example now.

Example. The following code plots a 8 bit pixel in a planar mode.

```
void pixel_p (unsigned char color, unsigned int x, unsigned int y) {

   unsigned int* fb        = (unsigned int*) 0xa0000;
   unsigned int offset     = y * pitch + (x/8);
   bankSwitch (offset >> 16);

   /* writes to bit mask register of graphics controller to select plane */
   outportb(0x3CE,8);
   outportb(0x3CF,0x80 >> (x & 7));
   fb [bankOffset] = color;
}
```

3. SuperVGA Interface

SuperVGA refers to the class of display hardware that includes SuperVGA (SVGA), and typically also XGA, SXGA, SXGA+, UXGA, QXGA, QSXGA, and other standards that support resolutions to 2560x2048 and higher. SuperVGA began as a set of extensions by manufacturers to VGA and adopted into separate SuperVGA standards that we have today. This means that **there is no all encompassing standard for SuperVGA devices.** Every SuperVGA card is different and provide different hardware and firmware interfaces. Thus we have a problem. How could we support SuperVGA if there is no standard way of using it? The problem is compounded by the fact that most manufacturers do not release the technical specifications for the cards. Instead they write black box drivers for different operating systems such as Windows and Linux.

Thus, in order for us to support SuperVGA devices we have only two options.

1. Write a device driver for each type of display device we want to support.
2. Use Vesa Bios Extensions (VBE)

The first option is hard because we need to have the specification to work off of. For devices where we cannot get any specification we have to rely on reverse engineering. The SuperVGA "card" we will look at is the one used in the Bochs emulator. We chose this one because anyone that uses Bochs could use it. The code for the driver will be Bochs specific however.

The second option is by using **Vesa Bios Extensions (VBE)**. VBE is a standard set of BIOS interrupt extensions for SuperVGA hardware. VBE is very simple to work with and creates a single standard interface for working with SuperVGA. It does however require real mode or v8086 mode and not all machines support VBE since it uses BIOS interrupts and is an extension.

3.1. Vesa Bios Extensions (VBE) Firmware Interface

Vesa Bios Extensions (VBE) defines a set of BIOS interrupt services for working with SuperVGA modes. VBE is defined in the **Vesa Bios Extensions (VBE) Core Standard**. (Please reference [5].) Kernel mode software could call the BIOS services in real mode or v8086 mode.

VBE Mode numbers

VBE defines a set of **mode numbers** in a similar way the video BIOS does. A mode number has the following format.

VBE Mode Number							
15	14	13	12	11	10	9	Bits 0...8
DM	LFB	Reserved, set to 0					Mode

Mode number is the actual mode number. If bit 8 is set, it is a standard VESA defined mode (more on this later.)

LFB selects the mode as a **Linear Frame Buffer (LFB)** or **Bank Switching** mode. If 0, it uses bank switching and uses the standard VGA frame buffer. If 1, it uses a linear frame buffer.

DM selects whether or not to clear display memory when setting the mode. If 0, display memory is cleared. If 1, display memory is not cleared.

VESA defined modes are a standard set of video mode numbers that BIOS venders are recommended to support. The following table lists the graphical modes.

Mode	Resolution	Color depth	Mode	Resolution	Color

					depth
100h	640x480	256	113h	800x600	32K
101h	640x480	256	114h	800x600	64K
102h	800x600	16	115h	800x600	16.8M
103h	800x600	256	116h	1024x768	32K
10Dh	320x200	32K	117h	1024x768	64K
10Eh	320x200	64K	118h	1024x768	16.8M
10Fh	320x200	16.8M	119h	1280x1024	32K
110h	640x480	32K	11Ah	1280x1024	64K
111h	640x480	64K	11Bh	1280x1024	16.8M
112h	640x480	16.8M			

Mode numbers beyond 11Bh can also be defined but are not standard. Thus there can be some support for even higher resolution modes.

Example. The VBE mode number **113h** selects **800x600x32K color mode** that uses bank switching while the VBE mode **8113h** selects **800x600x32 color mode** that uses a linear frame buffer (LFB). Remember the mode number format!

3.2. VBE Services

We now turn our attention to the VBE BIOS services. Please keep in mind all of these services can be referenced in [5]. We will also provide samples in C. We are only going to look at the three most important services required to set a video mode and display memory access. Please reference the specification (which is one of the easier specifications to read) for other interrupts.

INT 0x10 Function 4F00h – Get VBE Controller Information

Input:

- AX=4F00h
- ES:DI=Pointer to VbeInfoBlock structure (See following example)

Output:

- AX=Status

Structure. **vbeInfoBlock structure** has the following format.

```
typedef struct _vbeInfoBlock {
          uint8_t   signature[4];     // "VESA"
          uint16_t  version;          // Either 0x0200 (VBE 2.0) or 0x0300 (VBE 3.0)
          uint32_t  oemString;        // Far pointer to OEM name
          uint8_t   capabilities[4];  // capabilities
          uint32_t  videoModesPtr;    // Far pointer to video mode list
          uint16_t  totalMemory;      // Memory size in 64K blocks
          uint16_t  oemSoftwareRev;
          uint32_t  oemVenderNamePtr;
          uint32_t  oemProductNamePtr;
          uint32_t oemProductRevPtr;
          uint8_t   reserved [222];
          uint8_t   oemData [256];
}vbeInfoBlock;
```

The **vbeInfoBlock.capabilities** field has the following format.

Capabilities			
Bits 31...3	2	1	0
Reserved	D2	D1	D0

D0: If 0, DAC is fixed at 6 bits per color. If 1, DAC width can be switched to 8 bits per color.

D1: If 0, controller supports standard VGA modes. If 1, controller does not support standard VGA modes.

D2: If 0, RAMDAC is in normal operation. If 1, the specification tells us to use the blank bit of function 9.

The following example demonstrates calling this interrupt.

Example. There really isn't much to say here. To call the interrupt, we call our **rm_bios** routine with the structure. **SEG** and **OFFSET** are macros that calculate the real mode **segment** and **offset** respectively. This is used to convert the 32 bit linear address into a 16 bit segment:offset address when giving it to the BIOS.

```
void vbe_get_descr (vbeInfoBlock* descr) {
          INTR in, out;
          /* sanity checks */
          if (!descr)
```

```
                        return;

                /* call BIOS */
                in.eax.val =0x4F00;
                in.es = SEG((unsigned int) descr);
                in.edi.val = OFFSET((unsigned int) descr);
                io_services (0x10, &in, &out);
}
```

INT 0x10 Function 4F01h – Get VBE Mode Info

Input:

- AX=4F01h
- CX = Mode number (recall the format of a VBE mode number!)
- ES:DI=Pointer to ModeInfoBlock structure (See following example)

Output:

- AX=Status

Structure. **ModeInfoBlock** has the following structure. Due to the size of the structure we will not be describing all of the members. Important members that may be described in later chapters are commented.

```
typedef struct _modeInfoBlock {
  uint16_t attributes;
  uint8_t windowA, windowB;
  uint16_t granularity;
  uint16_t windowSize;
  uint16_t segmentA, segmentB;
  uint32_t winFuncPtr;                    /* ptr to INT 0x10 Function 0x4F05 */
  uint16_t pitch;                         /* bytes per scan line */

  uint16_t resolutionX, resolutionY;      /* resolution */
  uint8_t wChar, yChar, planes, bpp, banks; /* number of banks */
  uint8_t memoryModel, bankSize, imagePages;
  uint8_t reserved0;

  uint8_t readMask, redPosition;          /* color masks */
  uint8_t greenMask, greenPosition;
  uint8_t blueMask, bluePosition;
  uint8_t reservedMask, reservedPosition;
  uint8_t directColorAttributes;

  uint32_t physbase;                      /* pointer to LFB in LFB modes */
  uint32_t offScreenMemOff;
  uint16_t offScreenMemSize;
  uint8_t  reserved1 [206];
}modeInfoBlock;
```

The following example demonstrates calling the interrupt.

Example. The following function uses the above interrupt for basic bank switching. Note the use of **SEG** and **OFFSET** to convert the 32 bit linear address of **out** to a 16 bit segment:offset far pointer.

```
void vbe_get_mode (int mode, modeInfoBlock* descr) {
    INTR in, out;

    /* sanity check */
    if (!descr)
      return;

    /* call BIOS */
    in.eax.val = 0x4F01;
    in.ecx.val = mode;
    in.es      = SEG   ((unsigned int)descr);
    in.edi.val = OFFSET((unsigned int)descr);
    io_services (0x10, &in, &out);
}
```

INT 0x10 Function 4F02h – Set VBE Mode

The final interrupt that we will look at allows us to set the display mode. This can be used to set any VGA and VBE defined SuperVGA mode as well as extended modes that are not standard.

Input:

- AX=4F02h
- BX=Mode number (remember the VBE mode format!)

Output:

- AX=Status

The following example sets a VBE mode.

Example. The following function uses the above interrupt to set a VBE mode.

```
void vbe_set_mode (int mode) {
    /* call BIOS */
    INTR in, out;
    in.eax.val = 0x4F02;
    in.ebx.val = mode;
    io_services (0x10, &in, &out);
}
```

Calling **vbe_set_mode** allows us to set any VBE defined mode and yields us the following result.

Demo running in SuperVGA Mode 118h (1024x768)

3.3. VBE Display Memory

VBE supports two standard approaches used in SuperVGA for display memory: **Linear Frame Buffer (LFB)** and **bank switching**. You may recall that we can try to set any video mode supported by VBE with both of these. These are just different ways of accessing display memory. We will look at how to work with both types of modes in this section.

Linear Frame Buffer (LFB) Modes

In LFB modes, all of video memory is mapped into the **physical address space** typically at a high address in the 3GB-4GB range (assuming IA32 architecture.) Display memory is linear like an array in C. We can read or write to it by just reading or writing from a pointer and can access all of video memory. In VBE, we can get this pointer by calling **INT 0x10 Function 4F01h** to get the **modeInfoBlock** structure of the current mode. The pointer to display then is just at **modeInfoBlock.physbase**.

Example. The following C code plots a 16 bit **red-green-blue (RGB)** pixel to display.

```
void pixel_16RGB (unsigned short color, unsigned short x, unsigned short y) {
    unsigned short* fb = (unsigned short*) _modeInfo.physBasePtr;
    unsigned short offset = x + y * (_modeInfo.bytesPerScanLine / 2);
    fb [offset] = color;
}
```

Display memory in LFB modes can only be accessed from protected or long modes due to the amount of display memory mapped. In real mode the only way of accessing display memory of high resolution modes is using **bank switching**.

Bank switching

Modes that use **bank switching** always have 64K of display memory mapped at the VGA memory space at **0xa0000** physical address. This 64k block of display memory is called a **bank**. The software could only access a single bank at a time and so cannot access all of display memory. **The software has to switch banks when needed to access all of display memory.**

In other words, Bank 0 refers to bytes 0-64K of display memory, Bank 1 refers to bytes at 64K-128K of display memory, and so on. So by switching between banks the software can access all of display memory (no matter how high of resolution) through this 64K "window".

Example. The following code plots an 8 bit RGB pixel in a bank switching mode.

```
void pixel_8RGB (unsigned char color, unsigned short x, unsigned short y) {
    unsigned char* fb = (unsigned char*) 0xa0000;
    unsigned int offset = x + (long)y * _modeInfo.bytesPerScanLine;
    vbe_bankSwitch (offset >> 16);
    fb [offset & 0xffff] = color;
}
```

Bank switching modes are typically more slower because they require more calculations. That is, the software not only needs to calculate the location of a pixel on screen but also its offset within a bank and the bank to switch to. This same calculations apply to reading from display memory. Bank switching modes however can be used in real mode or v86 mode since they only map 64K of display memory.

Under VBE, software could switch banks by calling **INT 0x10 Function 0x4F05 – Display Window Control**. It can also be called directly (as is recommended by the VBE specification) by calling **INT 0x10 Function 0x4F01 – Get VBE Mode Information** and calling **VbeModeInfo.WinFuncPtr**.

INT 0x10 Function 4F05h – Display Window Control

Sets or gets the display bank. Can also be called from **VbeModeInfo.WinFuncPtr**.

Input:

- AX=4F05h
- BH=0 (Set memory window), 1 (Get memory window)
- BL=0 (Window A), 1 (Window B)
- DX=Window number in window granularity units. Only used when BH=0 (Set function)

Output:

- AX=Status
- DX=Window number in granularity units. Only used when BH=1 (Get function)

Example. The following function uses the above interrupt for basic bank switching. Notice that we call the interrupt twice for window A and window B. This is because, according to the standard, some VBE implementations might have separate read and write windows.

```
void vbe_bankSwitch (int bank) {
    INTR in, out;

    bank <<= bankShift;

    in.eax.val=0x4F05;
    in.ebx.val = 0;                 /* BH=0 (Set memory window) BL=0 (Window A) */
    in.edx.val = bank;
    io_services (0x10, &in, &out); /* call BIOS */

    in.eax.val=0x4F05;
    in.ebx.val = 1;                 /* BH=0 (Set memory window) BL=1 (Window A) */
    in.edx.val = bank;
    io_services (0x10, &in, &out); /* call BIOS */
}
```

4. Bochs VBE Interface

The Bochs emulator provides an alternative method of setting VBE modes directly without having to call the firmware. Although the support is limited, it does provide us a semi-portable method (only requires a compatible Bochs emulator) for working with high resolution graphics.

Example. The following functions can be used to set any VBE mode supported by Bochs.

```
#define VBE_DISPI_IOPORT_INDEX 0x01CE
#define VBE_DISPI_IOPORT_DATA  0x01CF
#define VBE_DISPI_INDEX_ID              0x0
#define VBE_DISPI_INDEX_XRES            0x1
#define VBE_DISPI_INDEX_YRES            0x2
#define VBE_DISPI_INDEX_BPP             0x3
#define VBE_DISPI_INDEX_ENABLE          0x4
#define VBE_DISPI_INDEX_BANK            0x5
#define VBE_DISPI_INDEX_VIRT_WIDTH      0x6
#define VBE_DISPI_INDEX_VIRT_HEIGHT     0x7
#define VBE_DISPI_INDEX_X_OFFSET        0x8
#define VBE_DISPI_INDEX_Y_OFFSET        0x9

#define VBE_DISPI_DISABLED              0x00
#define VBE_DISPI_ENABLED               0x01
#define VBE_DISPI_GETCAPS               0x02
#define VBE_DISPI_8BIT_DAC              0x20
#define VBE_DISPI_LFB_ENABLED           0x40
#define VBE_DISPI_NOCLEARMEM            0x80

void BlBochsVbeWrite (uint16_t index, uint16_t value) {

    WRITE_PORT_USHORT(VBE_DISPI_IOPORT_INDEX, index);
    WRITE_PORT_USHORT(VBE_DISPI_IOPORT_DATA, value);
}

void BlBochsSetMode (uint16_t xres, uint16_t yres, uint16_t bpp) {

    BlBochsVbeWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_DISABLED);
    BlBochsVbeWrite (VBE_DISPI_INDEX_XRES, xres);
    BlBochsVbeWrite (VBE_DISPI_INDEX_YRES, yres);
    BlBochsVbeWrite (VBE_DISPI_INDEX_BPP, bpp);
```

```
    BlBochsVbeWrite (VBE_DISPI_INDEX_ENABLE, VBE_DISPI_ENABLED | VBE_DISPI_LFB_ENABLED);
}
```

Recall that there is no standard Super VGA hardware interface. That is, this method is specific to Bochs and may not work in other environments or platforms. Because we assume most readers will be using Bochs as the primary emulator, this is the method we will use in order to maintain the highest form of compatibility. This method is also the simplest.

For higher compatibility however, it is recommended to use **virtual 8086** mode to utilize VBE services directly.

5. Demos

There are multiple demos that are planned for release for this article. These are a VGA BIOS demo, VBE SuperVGA demo, and VGA hardware demo.

6. Conclusion

We have covered a lot in this chapter including an introduction to the VGA BIOS, VGA hardware, SuperVGA, and VESA BIOS Extensions. We now have covered all of the material needed for switching into low and high resolution graphics modes that will be needed in later chapters. The next few chapters will be focused on the actual graphics rendering and pipeline with possibly some SuperVGA bits thrown in. We might begin with basic primitives and graphics rendering in the next article and transformations.

The method we will be using in later demos is the last one presented here, the Bochs VBE Interface. This allows us to use high resolution LFB modes for later articles with the assumption that most readers will be using Bochs. Any new code however added in later articles will be hardware interface independent for the most part; that is, you can use any method that you want to set the display mode and can still follow the next few articles that will focus more on graphics.

Until next time,

~Mike ();

OS Development Series Editor

6. Resources

The following links were referenced to provide more through and accurate information. Please reference them for additional information.

[1] http://en.wikipedia.org/wiki/Video_Graphics_Array

[2] http://www.osdever.net/FreeVGA/vga/vga.htm

[3] http://wiki.osdev.org/VGA_Hardware

[4] Programmer's Guide to the EGA, VGA, and Super VGA Cards (3rd Edition)

[5] VESA Bios Extensions (VBE) 2.0 Standard

Home

Operating Systems Development Series

Operating Systems Development - Multiboot

by Mike, 2010

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

This tutorial covers the multiboot standard and how to develop a multiboot-complient operating system. While the series goes into the multiboot structure, there is more to creating a multiboot complient system. By your system being multiboot complient, it will be capable of being loaded by any multiboot complient boot loader. Cool, huh? This means any multi boot complient bootloader can boot your OS.

Multiboot Specification

Abstract

The multiboot standard was originally created in 1995 and is overseen by the **Free Software Foundation**. They provide a written specification that defines a standard way to allow **multi-booting**. **Multi-booting** allows a computer system to install, and run, multiple operating systems and system environments. A **dual-boot** computer system is an example of multibooting, with two operating systems installed.

Multibooting is made possible by another, unofficial software trick: **Partitioning**. **Partitioning** creates the effect of multiple logical disks on one physical disk. Partitioning separates the storage on a storage medium (typically a hard disk) for different uses. For example, the first partition can be from sector 0 to sector *n* and contain an NTFS formatted Windows operating system. The other partitions can be of any file system - containing data or another operating system software.

Because partitioning is a software trick, it is up to the boot loader to be able to detect these partitions by reading the software **Partition Table** and, typically, display a list of the partitions that contain an operating system to boot. This is the **boot menu**. The Multiboot Specification defines the state of the computer when the bootloader transfers control to an operating system and how data is passed to the operating system.

The multiboot standard can be used on disks that do not support multibooting as well. This means, if your multi-boot complient bootloader can boot from a floppy disk, you can make your floppy disk OS boot from it.

Operating System Image

Typical bootloaders can be configured to boot different types of operating system images. Typically this is the Kernel or another OS Loading program. The multiboot specification does not provide details on the format of the image. Because of this, you can use any format that you want - flat binary, ELF, or even PE files.

However, this file requires an additional header - the **Multiboot Header**. This header must be located somewhere within the first 8k of your image and aligned on a dword (32 bit) boundery. Any multiboot complient bootloader will be able to find this header and obtain information from it. This is how the boot loader will know how to load and execute your image.

Here is the structure format:

```
typedef struct _MULTIBOOT_INFO {

   uint32_t magic;       //all required...
   uint32_t flags;
```

```
    uint32_t checksum;
    uint32_t headerAddr;  //all optional, set if bit 16 in flags is set...
    uint32_t loadAddr;
    uint32_t loadEndAddr;
    uint32_t bssEndAddr;
    uint32_t entryPoint;
    uint32_t modType;     //all optional, set if bit 2 in flags is set...
    uint32_t width;
    uint32_t height;
    uint32_t depth;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;
```

You should make sure no padding is added. In MSVC, this can be done by adding a **#pragma pack (push, 1)** and **#pragma pack(pop,1)** around the structure declaration above.

The above is the only structure that you need to get your OS booted by a multi boot complient bootloader, such as GRUB. Lets look at the members:

- **magic:** must always be 0x1BADB002
- **flags:**
  - **Bit 0**
    - 0: All boot modules and OS image must be aligned in page (4k) bounderies.
  - **Bit 1**
    - 1: Boot loader must pass memory information to the operating system.
  - **Bit 2**
    - 1: Boot loader must pass the video mode table to the operating system.
  - **Bit 16**
    - 1: Offsets 12-28 of the multiboot header are valid. (That is, members header_addr through entry_addr in the multiboot header are valid.) If this bit is set, the boot loader will use these values instead of parsing the image format and obtaining the values from it. Multiboot complient bootloaders can provide support for native executable file formats, such as ELF or PE that it can load.
- **checksum:** This must be a value that which, when added to **magic** and **flags** must be a 32 bit unsigned sum of 0.
- **headerAddr:** Address of multiboot header
- **loadAddr:** Base address to load to
- **loadEndAddr:** End load address. If 0, bootloader assumes the end is the end of the OS image file
- **bssEndAddr:** End of BSS segment. Bootloader null's this segment. If 0, no BSS segment is assumed
- **entryPoint:** Address of entry point function. Yes, thats right, the entry point of your operating system
- **modType:**
  - 0: Linear graphics mode
  - 1: EGA Standard text mode
  - Everything else is reserved
- **width:** width of display in text columns or pixels. If 0, the bootloader assumes no preference
- **height:** height of display in text columns or pixels. If 0, the bootloader assumes no preference
- **depth:** Number of **Bits Per Pixels (BPP)** in a graphics mode. If 0, the bootloader assumes no preference

That is all there is to it. The boot loader can load and execute your operating system in two ways: By loading and reading its executable image format (ELF and PE are examples) or by using the information found in this structure.

The boot loader looks for this structure in your image. Because of this, you need to fill out and create this structure.

Implementing the Multi boot Header

There are different solutions for implimenting the multiboot header into your operating system. Different solutions for different toolchains. Lets look at some of them.

Visual C++ 2005, 2008

This is a recent trick I discovered and posted on another forum. It uses some extensions provided by Microsoft Visual C++ to define the header in your kernel.

We first declare the structure, making sure there is no extra padding:

```
#pragma pack (push, 1)

/**
*   Multiboot structure
*/
typedef struct _MULTIBOOT_INFO {

   uint32_t magic;
   uint32_t flags;
   uint32_t checksum;
   uint32_t headerAddr;
   uint32_t loadAddr;
   uint32_t loadEndAddr;
   uint32_t bssEndAddr;
   uint32_t entryPoint;
   uint32_t modType;
   uint32_t width;
   uint32_t height;
   uint32_t depth;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;

#pragma pack(pop,1)
```

Now all that we need to do is define this structure somewhere. Remember that this header must be defined on a dword (32 bit) boundery and within the first 8K of your kernel? This trick uses section alignment to insure the proper alignment of the structure. We set up the section alignment in the **Linker Options** of the IDE and it is guaranteed to be dword aligned. So, all we need to do is create a new program section and define the structure in it. Neat, huh?

Lets do that now:

```
//! Bad example:
#pragma section(".text")
__declspec(allocate(".text"))
MULTIBOOT_INFO _MultibootInfo = {

   MULTIBOOT_HEADER_MAGIC,
   MULTIBOOT_HEADER_FLAGS,
   CHECKSUM,
   HEADER_ADDRESS,
   LOADBASE,
   0, //load end address
   0, //bss end address
   KeStartup
};
```

This works but is problimatic. This allocates the structure in the .text section, but *where?* This is going to be a problem - Multiboot specification requires the structure be located in the first 8K of the image, but MSVC is still free to place it outside the 8K region.

To fix this problem, we must use the section naming convention. The section naming conventions used in MSVC follow the format **name$loc** where **name** is the name of the section, and **loc** is an alpha-numeric value that represents where, in the section, it represents. Its in alphanumeric order: **section$a** is first, **section$b** is second and so on. So, by using **.text$0** we are representing the beginning of the .text segment. But of course, just replacing the above **.text** to **.text$a** wont work - My, or my no, that would be too easy. :)

Instead, we define our own section - lets call it **.a$0**. We can create this as a code segment and merge it into the .text section:

```
//! Complete example
#pragma code_seg(".a$0")
__declspec(allocate(".a$0"))
MULTIBOOT_INFO _MultibootInfo = {

   MULTIBOOT_HEADER_MAGIC,
   MULTIBOOT_HEADER_FLAGS,
```

```
   CHECKSUM,
   HEADER_ADDRESS,
   LOADBASE,
   0, //load end address
   0, //bss end address
   KeStartup
};

#pragma comment(linker, "/merge:.text=.a")
```

Thats all that there is to it. **KeStartup** is your entry point function, **OADBASE** is the base address of your kernel (like 1MB for example), **HEADER_ADDRESS** is the address of the multiboot header (which happens to be **LOADBASE+0x400** do to .text always starting at offset 0x400), magic is **0x1BADB002**, flags of **0x00010003** and the checksum being **-(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)**.

Here is the complete example:

```
#pragma pack (push, 1)

/**
*   Multiboot structure
*/
typedef struct _MULTIBOOT_INFO {

   uint32_t magic;
   uint32_t flags;
   uint32_t checksum;
   uint32_t headerAddr;
   uint32_t loadAddr;
   uint32_t loadEndAddr;
   uint32_t bssEndAddr;
   uint32_t entryPoint;
   uint32_t modType;
   uint32_t width;
   uint32_t height;
   uint32_t depth;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;

#pragma pack(pop,1)

/**
*   Kernel entry
*/
void KeStartup ( PMULTIBOOT_INFO* loaderBlock ) {

   __halt ();
}

//! loading address
#define LOADBASE                    0x100000

//! header offset will always be this
#define   ALIGN                           0x400
#define   HEADER_ADDRESS         LOADBASE+ALIGN

#define MULTIBOOT_HEADER_MAGIC        0x1BADB002
#define MULTIBOOT_HEADER_FLAGS        0x00010003
#define STACK_SIZE                    0x4000
#define CHECKSUM                      -(MULTIBOOT_HEADER_MAGIC + MULTIBOOT_HEADER_FLAGS)

#pragma code_seg(".a$0")
__declspec(allocate(".a$0"))
MULTIBOOT_INFO _MultibootInfo = {

   MULTIBOOT_HEADER_MAGIC,
   MULTIBOOT_HEADER_FLAGS,
   CHECKSUM,
   HEADER_ADDRESS,
   LOADBASE,
   0, //load end address
   0, //bss end address
   KeStartup
};
```

```
#pragma comment(linker, "/merge:.text=.a")
```

Assuming this kernel has the base address of 1MB, and is compiled with Visual C++ to produce a valid PE executable, this should be bootable by any multiboot complient bootloader.

Machine State

When the bootloader executes our operating system, the registers must be the following values:

- **EAX** - Magic Number. Must be 0x2BADB002. This will indicate to the kernel that our boot loader is multiboot standard
- **EBX** - Containes the physical address of the Multiboot information structure
- **CS** - Must be a 32-bit read/execute code segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact value is undefined.
- **DS,ES,FS,GS,SS** - Must be a 32-bit read/write data segment with an offset of `0' and a limit of `0xFFFFFFFF'. The exact values are all undefined.
- **A20 gate** must be enabled
- **CR0** - Bit 31 (PG) bit must be cleared (paging disabled) and Bit 0 (PE) bit must be set (Protected Mode enabled). Other bits undefined

All other registers are undefined. Most of this is already done in our existing boot loader. The only additional two things we must add are for the EAX register and EBX. The most important one for us is stored in EBX. This will contain the physical address of the multiboot information structure. Lets take a look!

Multi boot Information Structure

Now that our operating system is being booted by the boot loader, whats next? Multiboot complient boot loaders also creates an information structure providing information to the operating system. These are passed by a pointer to the structure in the **EBX** register.

This is possibly one of the most important structures contained in the multiboot specification. The information in this structure is passed to the kernel from the EBX register, **This allows a standard way for the boot loader to pass information to the kernel.**

This is a fairly big structure but isnt to bad. Not all of these members are required. The specification states that the operating system must use the flags member to determin what members in the structure exist and what do not.

Here is the entire structure format. Simular to the multi-boot header structure, it is recommended to insure there is no padding added.

```
typedef struct _MULTIBOOT_INFO {

        uint32_t flags;                 //required
        uint32_t memLower;              //if bit 0 in flags are set
        uint32_t memUpper;              //if bit 0 in flags are set
        uint32_t bootDevice;            //if bit 1 in flags are set
        uint32_t commandLine;           //if bit 2 in flags are set
        uint32_t moduleCount;           //if bit 3 in flags are set
        uint32_t moduleAddress;         //if bit 3 in flags are set
        uint32_t syms[4];               //if bits 4 or 5 in flags are set
        uint32_t memMapLength;          //if bit 6 in flags is set
        uint32_t memMapAddress;         //if bit 6 in flags is set
        uint32_t drivesLength;          //if bit 7 in flags is set
        uint32_t drivesAddress;         //if bit 7 in flags is set
        uint32_t configTable;           //if bit 8 in flags is set
        uint32_t apmTable;              //if bit 9 in flags is set
        uint32_t vbeControlInfo;        //if bit 10 in flags is set
        uint32_t vbeModeInfo;           //if bit 11 in flags is set
        uint32_t vbeMode;               // all vbe_* set if bit 12 in flags are set
        uint32_t vbeInterfaceSeg;
        uint32_t vbeInterfaceOff;
        uint32_t vbeInterfaceLength;

}MULTIBOOT_INFO, *PMULTIBOOT_INFO;
```

This structure isnt as complex as it looks. If the corrosponding bit in the flags member is set, it means that the members (shown above) are valid. Because of this, **flags** is, technically, the only required member, all other members are optional.

Lets look at the members here:

- **memLow, memUpper:** Amount of low and upper memory in KB. Low memory starts at 0, upper memory starts at 1MB.
- **bootDevice:** Boot device (see below)
- **commandLine:** Pointer to C-string containing your kernel command line
- **moduleCount:** The number of additional boot modules that were loaded by the boot loader
- **moduleAddress:** address of first **module structure** (see below)
- **syms:** Location of symbol table. See below
- **memMapLength:** Number of entries in system memory map
- **memMapAddress:** Address of memory map
- **drivesLength, drivesAddress:** see below
- **configTable:** Address of BIOS ROM config table (returned from GET CONFIGUATION BIOS INT call)
- **apmTable:** Address of **Advanced Power Management (APM)** table
- **vbeControlInfo, VbevbeModeInfo:** Address of **Video Bios Extensions (VBE)** structures.
- **vbeMode:** VBE mode
- **vbeInterfaceSeg, vbeInterfaceOff, vbeInterfaceLength:** Used to access VBE 2.0 protected mode interface

This chapter does not go over VBE nor APM so we wont cover them here. Memory map information has been described in Chapter 17, including the format of the **System Memory Map**.

The ROM configuation for **configTable** is the table obtained from BIOS INT 0x15 Function 0xC0.

Thats all there is to it. This structure isnt that bad :) There are a few members we havnt looked at though: bootDevice, moduleAddress, syms, drivesLength, and drivesAddress. Lets look at those in detail.

bootDevice

The **bootDevice** member follows the format:

- 1st word: BIOS drive number
- 2nd, 3rd, 4th words: Partition

The BIOS drive number is the number used by the BIOS INT 0x13 services to represent the drive. The other words reoresent the partitions: Words 2,3, and 4 represent Partition 1,2,and 3. Partition 1 is the top level partition, partition 2 is the sub-partition in that partition and so on. Unused partitions are marked as 0xFF.

moduleAddress

This is a pointer to the first module structure. A module structure entry follows the format:

```
typedef struct _MODULE_ENTRY {

        uint32_t moduleStart;
        uint32_t moduleEnd;
        char     string[8];

}MODULE_ENTRY, *PMODULE_ENTRY;
```

moduleStart and **moduleEnd** contain the start and end addresses of the loaded module. "string" represents that module, typically can be a command line or path name or 0 if there is none.

drivesLength, drivesAddress

The **drivesLength** member contains the size of all of the drive structures. **drivesAddress** contains a pointer to the first drive structure. A drive structure entry has the format:

```
typedef struct _DRIVE_ENTRY {

        uint32_t size;                  //size of structure
```

```
        uint8_t driveNumber;
        uint8_t driveMode;
        uint16_t driveCylinders;
        uint8_t driveHeads;
        uint8_t driveSectors;
        uint8_t ports [0];              //can be any number of elements

}DRIVE_ENTRY, *PDRIVE_ENTRY;
```

Lets take a closer look at each member:

- **driveNumber:** Number used by BIOS
- **driveMode:**
  - 0: CHS
  - 1: LBA
- **driveCylinders,driveHeads,driveSectors:** Drive Geometry
- **ports:** contain a list of I/O port numbers used by the BIOS for drive access, terminated by 0

Thats all there is to this structure. Only one more member to cover, that strange **syms** member...lets take a look!

syms

The **syms** member is declared in the structure in this chapter as **uint32_t syms[4]** but that is not entirely true. It is actually several members that occupy those bytes that follow the members:

- syms[0] = uint32_t sym_num
- syms[1] = uint32_t sym_size
- syms[2] = uint32_t sym_addr
- syms[3] = uint32_t sym_shndx

While the specifications state that any format for the operating system image can be used (Such as ELF, PE, flat binary, or anything), this one is specific to ELF formats only. Technically the system image (like the kernel) is able to parse itself to obtain symbolic information. However the miltiboot standard allows the boot loader to pass ELF-specific symbolic information to the operating system as well through these members. **sym_num** is the number of symbol entries in the ELF section header. **size** indicates the size of each entry, **addr** contains the address of the symbol table in the ELF binary.

Conclusion

Thats all that there is to the Multi boot standard! Technically all that you need is to define the **Multiboot Header** properly in order to get your system booted by any multiboot complient bootloader. However, you can use the **multiboot information** structure to obtain information that is normally obtained at boot time.

If you would like to support multi boot in the series, you must insure your kernel is loaded at a **physical** address not a virtual one. This is because paging is disabled when a multiboot complient bootloader passes control to your kernel. A good typical address is 1MB, which can be loaded by alot of bootloaders, such as GRUB. You can, of course, enable paging and use it later on of course :)

Until next time,

~Mike
BrokenThorn Entertainment. Currently developing DoE and the Neptune Operating System Software Suite.

Questions or comments? Feel free to Contact me.

Home

Operating System Development Series

IA32 Machine Language

by Mike, 2011

Introduction

This chapter covers IA32 machine language programming. The information provided here is for information purposes only and is not needed for the development of a basic operating system or executive software. Understanding the instruction format for the IA32 (and IA64) instructions can help debugging improperly assembled instructions, v86 monitors that are required for supporting v8086 mode, emulating instructions (which is required for emulating certain FPU instructions or when developing assemblers, emulators, virtual machines, and some other types of software), and when developing certain system software like debuggers and compilers.

This chapter is also for testing a new editor being used to write the new chapters that should help improve formatting and resolve spelling errors. If this test is successful, all of the new and earlier chapters will be updated to reflect the new format. Please send any feedback if you encounter any errors.

Machine Language Overview

Machine language, also known as **machine code, native code,** and **byte code,** is the set of raw instructions and data that can be executed by a **central processing unit (CPU)**. It allows a CPU to interpret a certain set of byte sequences as an "instruction" to perform a task. These tasks are very small, such as copying small amounts of data or arithmetic. The act of building a byte sequence that represents a CPU instruction is known as **coding**. The definition of **coding** has evolved as programming languages evolved. Originally the term referred to the actual coding of the byte sequence for an instruction; today it applies to many forms of programming in second, third, and fourth generation programming languages. Computer **programs**, also known as **software,** is the collection of machine code and data that performs a complicated task, such as word processing or playing Halo®. Machine language is often interpreted by popular media as a "series of 1's and 0's". This is an accurate description—to an extent.

Digital Logic

Digital logic is a field of electronics that utilizes **logic gates** that allows the electronics to make decisions. Some examples of logic gates include **AND gates, OR gates, NOR gates, NAND gates, NOT gates** and **XOR gates.** These gates reflect their binary operations: AND gates perform a binary AND, XOR performs a binary XOR, and so on. In order for these gates to be meaningful, a standard needed to be adopted in order to make sense of what is **true** and **false**. For example, AND gates only make sense if there are two inputs and one output. The two inputs are the two items to test for equality: if either one is false, the output is false else the output is true. The standard is to define a line with low electrical current as **false** and a line with higher electrical current as **true**. This is what connects the binary number system to digital logic electronics. In the binary number system, **0** is often denoted as **false** and **1** is often **true**. Machine language is often represented in binary because of its tight connection with the CPU instruction decoding mechanism and how its stored in RAM for instruction fetching.

Program loading

Programs are loaded into memory by the operating system, executive, or firmware. The IA32 and IA64 family of CPU's can execute programs from **Read Only Memory (ROM)** and **Random Access Memory (RAM)**. This is made possible through a common **system bus** and **physical address space (PAS)** shared by firmware and program images. Because firmware and program images can both be executed directly by the CPU cores, they utilize the same machine language byte code. Machine language is different then **microcode** (see chapter 7) that firmware might use, however, the actual firmware is still machine language.

Assembly language

Assembly language is a **second generation programming language**. It allows the programmer to write a program in a well defined language that uses **mnemonics** to aid the programmers in developing the software. For example, **MOV** is a common mnemonic common in a lot of assembly languages for different architectures. **MOV** represents an instruction that copies data from a source to a destination. It is also an example of a **data movement instruction**.

The mnemonics gave a symbolic name to the instructions and instruction forms, allowing each assembly language instruction to be translated to a single (in some cases, several possible) machine language byte sequences. The program that translates assembly language instructions into machine language is known as an **assembler**. Assemblers are sometimes incorrectly called **compilers**.

Machine Instruction overview

A **machine instruction** is a single byte sequence that performs a specific task for the CPU. A set of machine instructions has been previously defined as a **machine language**. Machine instructions are defined by a CPU manufacturer in an **instruction set** that documents all of the CPU instructions the manufacturer implemented. Instruction sets also typically include suggestive assembly language mnemonics for assembler developers to use. Instruction sets are documented in CPU specifications.

The CPU manufacture implements the machine instructions supported by a particular CPU and how the CPU interprets each instruction. This allows the CPU to "execute" machine instructions that the manufacture intended. Bugs in the CPU hardware or firmware, however, can cause the CPU to "execute" instructions that are not valid instructions. These are **undocumented instructions**. Some assemblers might define mnemonics to undocumented instructions that are well known. Some undocumented instructions that have had benefits have later become documented as real instructions. Some instructions might be left undocumented for manufacturer testing use only, such as the IA32 **LOADALL** instruction (this bug has since been fixed.) Some instructions might also have bad effects, such as halting the system or damaging the CPU (these are known as **Halt and Catch Fire (HCF)** instructions.)

There is an instruction set for every CPU architecture. Due to the evolving nature of the software industry, certain trends in instruction sets have become common. Understanding these trends can help with understanding IA32 machine language.

CISC and RISC

Instruction sets typically fall in two categories: **Complex Instruction Set Computing (CISC)** and **Reduced Instruction Set Computing (RISC)**. Examples of RISC include PPC and ARM architectures. An example of CISC is the IA32 architecture. RISC architectures utilize a simplistic instruction set format over CISC. RISC architectures typically uses a standard encoding format for each instruction that allows each instruction to be of the same number of bytes. CISC architectures also follow a standard encoding format, but allows variable length instructions.

Operation code

An **Operating Code (OPCode)** is a single byte identifier that the CPU utilizes to determine the instruction type. For example, a MOV instruction has an opcode identifier that lets the CPU know information about the instruction, such as type (MOV). Many instruction sets use opcode to distinguish one instruction from another. Some instructions may have **multi-byte opcodes** and **extended opcodes**. This gives the instruction set more flexibility.

Addressing mode

An **Addressing Mode** defines a method for the CPU to be able to reference **addresses.** The addresses may be virtual or physical depending on the architecture. Instructions, such as **data movement instructions**, need to be able to tell the CPU how to reference addresses to obtain data. For example, many CPU's support a **direct addressing mode** that allows an instruction to tell the CPU to reference (read or write) data at a specific address. For example, in IA32 assembly language:

```
mov eax, dword [0xa0000]
```

This instruction tells the CPU to use the **direct addressing mode** to read from address 0xa0000 in the current address space. Another common addressing mode is **indirect addressing**, which allows an instruction to tell the CPU to reference data using a pointer. For example, in IA32 assembly language:

```
mov eax, [ebp]
```

This tells the CPU to read a dword from the address stored in the EBP register into the EAX register. There are many more addressing modes that architectures may utilize.

IA32 and IA64 Instruction encoding

We are now ready to begin looking at IA32 and IA64 machine instruction encoding. In order to save space, we will use IA64 to mean IA32 and IA64 instruction sets. IA32 is a subset of IA64 and thus shares a large part of the IA32 instruction set. The IA64 instruction set implements a CISC encoding. This means that each instruction follows a specific encoding structure and is variable in length. IA32 and IA64 instructions can range from 1 byte to 12 bytes in size.

Register codes

The CPU identifies internal registers by a numerical value. Many registers share the same code; the CPU decides what register to use based on the instruction being used and the current operating mode (real, protected, or long modes). The **operand size override** prefix is also used when determining what register to use. We will cover this prefix later on.

Register codes are used in the instruction encoding to let the CPU know what registers the instruction operates on. The registers use the following codes:

REX.r = 0

Code	0	1	2	3	4	5	6	7
No REX	AL	CL	DL	BL	AH	CH	DH	BH
REX	AL	CL	DL	BL	SPL	BPL	SIL	DIL
REG16	AX	CX	DX	BX	SP	BP	SI	DI
REG32	EAX	ECX	EDX	EBX	ESP	EBP	ESI	EDI
REG64	RAX	RCX	RDX	RBX	RSP	RBP	RSI	RDI
MM	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM0	XMM1	XMM2	XMM3	XMM4	XMM5	XMM6	XMM7
YMM	YMM0	YMM1	YMM2	YMM3	YMM4	YMM5	YMM6	YMM7
SSEG	ES	CS	SS	DS	ES	GS		
	CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7
	DR0	DR1	DR2	DR3	DR4	DR5	DR6	DR7

For example, in the instruction **mov bx, 0x5** we would store 3 as the register code for BX. The instruction **mov ss, ax** would require storing 2 as the register code for SS and 0 for the register code of AX. Different instructions utilize different types of registers so there will never be a conflict between needing to choose between multiple registers of the same code. For example, the instruction **mov REG16, IMM16** will always use a 16 bit general purpose register as an operand. For another example, the instruction **movups xmm, xmm/m128** will always take an XMM register only.

Long mode adds additional registers to this list. In order to support the above registers, long mode has a special flag set that allows instructions to select other registers using the same register codes. This is the **REX.r** field in the **REX prefix byte** that will be explained later on. When this bit is set, the register table will look like this:

REX.r=1

Code	0	1	2	3	4	5	6	7
No REX	R8B	R9B	R10B	R11B	R12B	R13B	R14B	R15B
REX	R8W	R9W	R10W	R11W	R12W	R13W	R14W	R15W
REG16	R8D	R9D	R10D	R11D	R12D	R13D	R14D	R15D
REG32	R8	R9	R10	R11	R12	R13	R14	R15
MM	MM0	MM1	MM2	MM3	MM4	MM5	MM6	MM7
XMM	XMM8	XMM9	XMM10	XMM11	XMM12	XMM13	XMM14	XMM15
YMM	YMM8	YMM9	YMM10	YMM11	YMM12	YMM13	YMM14	YMM15
SSEG	ES	CS	SS	DS	FS	GS		
	CR8	CR9	CR10	CR11	CR12	CR13	CR14	CR15
	DR8	DR9	DR10	DR11	DR12	DR13	DR14	DR15

Instruction Encoding

An IA64 instruction follows a well defined structure that originated from the 8085 CPU. Each instruction follows the following format:

Prefix bytes (0-4)	REX prefix (1)	Operation (0-3)	Mod R/M (1)	SIB (1)	Displacement (0-4)	Immediate (0-4)

For compactness, the number in parentheses is the number of bytes of the component. A number of 0 indicates that the byte is optional. For example, the **prefix bytes** can be from 0 to 4 bytes in an instruction. This means that an instruction can have 0, 1, 2, 3, or 4 prefix bytes. **The REX prefix is only valid in IA64 and long modes.** The only required field is an **operation code**. All other fields are optional and depend on if the instruction requires them. For example, the **INT (interrupt)** instruction requires an operation code and immediate byte while a **MOV** instruction might utilize all of the above fields.

To provide another example, lets take a look at the INT instruction in more detail. The INT instruction has a form:

```
INT imm8
```

where imm8 is an 8 bit immediate value and INT is the mnemonic for the operation code 0xCD. Knowing the format of the instruction encoding, we can encode a INT 5 instruction like this:

```
0xCD 0x05
```

The first byte, 0xCD, is the operation code and is in **brown**. Because prefix bytes are optional, and INT 5 does not require them, we do not need it. Mod R/M and SIB bytes are not needed either. Displacements are only used with memory a**ddressing modes** so the only other field that we need is the immediate field. The immediate field is a 0-4 byte field. We know to use it as a 1 byte field because of the instruction form INT imm8. The purpose of this example is to demonstrate that **certain fields are optional and not needed**; the fields that are needed by an instruction depends on the instruction. **The order of these fields never changes**. For example, note above how we chose to omit the fields that are not needed but kept the order of the fields: the operation code field comes before the immediate value field. In the next few sections, we will cover each of these fields in more detail.

Prefix fields

The **prefix bytes** allow an instruction to give more information to the CPU. For example, they allow the instruction to have the CPU lock the bus or to utilize a different segment register in a data movement instruction. Many of these prefixes have assembly language mnemonics. The prefix bytes are identified in 4 classes. **An instruction can use at most 1 prefix byte from each of the 4 classes**.

Class	1 prefix
0xF0	LOCK prefix
0xF2	REPNE, REPNZ prefix
0xF3	REP, REPZ, REPE prefix

Class	2 prefix
0x2E	CS Segment override
0x36	SS Segment override
0x3E	DS Segment override
0x26	ES Segment override
0x64	FS Segment override
0x65	GS Segment override

Class	3 prefix
0x66	Operand size override

Class	4 prefix
0x67	Address size override

We assume the reader knows IA32 assembly language so will omit describing these prefixes in detail. A machine instruction can only have 1 prefix byte from any of the 4 classes. Due to their being 4 classes, an instruction can have 0 to 4 prefix bytes. If an instruction attempts to use 2 or more prefix bytes from a single class, the CPU will throw an invalid instruction exception.

LOCK prefix

If the **LOCK** prefix is used on an instruction that does not support LOCK the CPU will trigger an invalid instruction exception. Some assemblers allow using LOCK on invalid instructions without giving a warning to the programmer. Due to this, we will present the list of valid instructions here.

The **LOCK** prefix can only be used on the following instructions: ADC, ADD, AND, BTC, BTR, BTS, CMPXCHG, CMPXCHG8B, CMPXCHG16B, DEC, INC, NEG, NOT, OR, SBB, SUB, XADD, XCHG, and XOR.

Operand size override

The **operand size override** allows the CPU to select between 16 bit and 32 bit operands. Assemblers typically allow the programmer to select a specific operand size indirectly using directives like **bits16** or **use32.** The IA32 and IA64 instruction sets provide two operand sizes: legacy 16 bit and a native size that is 32 bit. The native size depends on the processors current operation mode.

Operation mode	CS.d	REX.w	Native	Operand override
Real mode			16 bit	16 bit
V8086 mode			16 bit	16 bit
Protected mode	0		16 bit	32 bit
Protected mode	1		32 bit	16 bit
Long mode		0	32 bit	16 bit

For an example, lets look at the ADD AX/EAX, IMM16/IMM32 instruction. This instruction has operation code 0x05. In protected mode code, the CPU will interpret this as an ADD EAX, IMM32 instruction by default. However we can override the default behavior and copy a 16 bit immediate value by using an operand override prefix. We do this in assembly language like this:

```
add eax, 5 ; MOV EAX, IMM32
add ax, 5  ; MOV AX, IMM16
```

The first instruction will assemble to:

```
0x05 0x05 0x00 0x00 0x00
```

The second instruction will assemble to:

```
0x66 0x05 0x05 0x00
```

Notice the only differences between these two instructions are the following: (1) The first instruction uses a 32 bit immediate value and the second instruction uses a 16 bit immediate value (these are in **red**), and (2) the second instruction uses the operand override prefix (this is in **black**). This tells the CPU to use the 16 bit operand form. For completeness, the values in **brown** are the operation codes.

Address size override

The address size override prefix byte is very similar to the operand override prefix byte. Assemblers allow programmers to be able to select between address sizes by using keywords such as **byte ptr** and **dword ptr.** Due to the function being very similar to the operand override prefix, we will omit describing its purpose because it is the same but applies to address modes.

Operation mode	CS.d	REX.w	Native	Address override
Real mode			16 bit	16 bit
V8086 mode			16 bit	16 bit
Protected mode	0		16 bit	32 bit
Protected mode	1		32 bit	16 bit
Long mode		0	64 bit	32 bit
Long mode		1	64 bit	32 bit

For example, the instruction **mov eax, [0xa000]** when assembled in protected mode would not need an address size override. The assembler would treat 0xa000 as a 32 bit displacement. However, if we used **mov ax, word [0xa000]** the assembler would add an address size override prefix to the instruction to select the 16 bit address form.

REX prefix

The REX prefix enables certain 64 bit specific features. It has the following format:

```
| 0 | 1 | 0 | 0 | w | r | x | b |
+---+---+---+---+---+---+---+---+
7                               0
```

REX.w	Operand size. 0: Default, 1: 64 bit
REX.r	ModRM.reg extension
REX.x	SIB.index extension
REX.b	ModRM.rm extension

Prefix Order

The order of the prefix bytes when used in conjunction with other prefix bytes does not matter. For example, you can use **0xF3 0x2E** in the machine code to select REP and CS override. You can also use **0x2E 0xF3** to do the same thing.

Operation code field

The operation code field can be 1-3 bytes in length. All operation codes are unique; they identify the instruction to use and its operands. For example, the operation code 0 identifies the ADD REG/MEM8, REG8 instruction. The operation code 1 identifies an ADD REG/MEM16/MEM32, REG16/REG32 instruction. The IA32 and IA64 CPU manuals outline each instruction and their respective operation code.

Primary Opcode

The **primary opcode** is a single byte that is required in all instructions. It is the base of the operation code fields used to identify the instruction. The primary opcode can also take on one of the following formats depending on instruction.

```
|   |   |   |   |   |   | d | s | w |
+---+---+---+---+---+---+---+---+---+
7                               0

|   |   |   |   |   |     tttn      |
+---+---+---+---+---+---+---+---+---+
7                               0

|   |   |   |   |   | register ID|
+---+---+---+---+---+---+---+---+---+
```

```
7                                   0

|   |   |   |   |   |   |   mf  |
+---+---+---+---+---+---+---+---+
7                                   0
```

PO.w	Operand size
PO.s	Sign extend
PO.d	Direction
PO.tttn	Used on some FPU instructions
PO.mf	Memory format

Secondary OPCode

When the primary opcode byte is 0xf0, another byte follows called the secondary opcode byte. The secondary opcode then identifies different instructions and has the same functionality as above. These should be treated as two byte opcodes.

OPCode extension

Certain families of instructions has the same opcode but differ only by a special field called an **opcode extension**. This is a 3 bit extension that is stored in the Mod R/M.reg field. The Mod R/M byte will be explained in more detail in the next section.

Multi-byte OPCodes

The primary opcode field can be 1-3 bytes in length. Although most instructions only use 1 byte of the primary opcode field, there are some that can utilize all 3 bytes. All of these instructions also uses a secondary opcode byte (0xf0).

Mod R/M field

The Mod R/M (Register/Memory) field is used by instructions that require memory or register operands. The Mod R/M field has the following format.

```
| mod   |    reg    |    rm     |
+---+---+---+---+---+---+---+---+
7                               0
```

The Mod R/M field is slightly different depending on if the CPU is running in real, protected or long modes.

Real mode

ModRM.mod	00: [Memory] 01: [Memory+DISP8] 10: [Memory+DISP16] 11: Register
ModRM.reg	Register code
ModRM.rm	If ModRM.mod = 11: register code 000: [BX+SI] 001: [BX+DI] 010: [BP+SI] 011: [BP+DI] 100: [SI] 101: [DI] 110: [BP] or [DISP16] when ModRM.mod=0 111: [BX]

Protected and Long modes

ModRM.mod	00: [Memory] 01: [Memory+DISP8] 10: [Memory+DISP32] 11: Register
ModRM.reg	Register code
ModRM.rm	If ModRM.mod = 11: register code REX.b=0 / REX.b=1 000: [RAX] / 000: [R8] 001: [RCX] / 001: [R9] 010: [RDX] / 010: [R10] 011: [RBX] / 011: [R11] 100: [SIB] / 100: [SIB] 101: [RBP][DISP32] / 101: [DISP32] 110: [RSI] / 110: [R14] 111: [RDI] / 111: [R15]

The ModRM.mod field is combined with Mod.rm field to determine the addressing mode. For example, the instruction **mov ax, [0xa000]** would use (in real mode) ModRM.mod = 0 (Memory) and ModRM.rm = 6 (DISP16). ModRM.reg would contain the register code for AX. If we are to use **mov ax, [bx+0xa000]** instead, ModRM would be 2 (Memory+DISP16) and ModRM.rm would be 7. Assemblers would treat 0xa000 here as a DISP16 rather then a DISP8 due to 0xa000 being a word size displacement. We can select the DISP8 form by using an **address size override** prefix.

Looking at the above tables we can deduce that there is not many registers can be used for indirect addressing. For example, **[BP]** is not a valid addressing mode, but assemblers can assemble this fine in instructions like **mov ax, [bp]**. A common trick used by these assemblers is to set ModRM.mod = 1 (Memory+DISP8) and ModRM.rm = 6 (BP). In other words, the assemblers translate this into a **[BP+DISP8]** addressing mode, setting the displacement to 0. So **mov ax, [bp]** is assembled into **mov ax, [bp+0]**.

Protected and long modes introduce another addressing mode, **[SIB]** which gives more capabilities. SIB addressing modes can be combined with ModRM.rm modes. For example, in protected mode, **mov eax, [ebx+edi*2+0xa000]** would be translated to ModRM.mod = 2 (Memory+DISP32) and ModRM.rm = 4 (SIB byte). The SIB byte tells us how to extract the **edx+edi*2** in this instruction so will be explained in the next section.

Certain instructions utilize an **extended opcode** field. Extended opcodes are identifiers that are used with the **primary opcode** when identifying instructions. This is a 3 bit field and is stored in Mod R/M.reg field for these instructions. Instructions that use an extended opcode field might still use ModRM.rm and ModRM.mod to store a register operand or memory addressing mode.

SIB field

The **Scale Index Base (SIB)** byte **follows a Mod R/M byte only if Mod R/M.rm = 4 and the CPU is in protected or long modes**. The byte provides additional addressing modes to the IA32 and IA64 architectures. SIB addressing is combined with Mod R/M addressing in order to create a wide array of additional addressing modes.

```
| Scale |   Index   |   Base    |
+---+---+---+---+---+---+---+---+
7                               0
```

SIB.Scale	00: Factor 1 01: Factor 2 10: Factor 4 11: Factor 8
SIB.Index	Uses standard register codes If VSIB, uses VR register codes If REX.x = 1, uses 64 bit register codes If REX.x = 1 and VSIB, uses VR register codes
SIB.Base	Uses standard register codes If REX.b=1, uses 64 bit register codes

Despite the names of the register fields in the SIB byte, you can technically use any register code. For example, you can put an index register in SIB.Base.

Lets combine the SIB byte with Mod R/M again to demonstrate how they work together. Using our previous example, **mov eax, [ebx+edi*2+0xa000]**. We assume the CPU is running in protected mode for this example. EAX is the non-memory register, so that will be in Mod R/M.reg. We also have to set Mod R/M.mod = 2 to enable [Memory+DISP32] and Mod R/M.rm = 4 [SIB byte]. **ebx** is our base register, **edi** is our index register. The **register code** for EBX is 3 and the **register code** for EDI is 7. Using this, we can set SIB.index = 7 and SIB.base = 3. The scale factor, **2** goes into SIB.scale.

Putting this together, we have a Mod R/M byte of **10 000 100 binary** and an SIB byte of **10 111 011 binary**. Knowing we have a 32 bit displacement of **0xa000** and the operation code being **0x89**, we can translate our example instruction into:

```
0x89 0x84 0xbb 0x00 0xa0 0x00 0x00
```

This would be the correct translation of **mov eax, [ebx+edi*2+0xa000]**. To ease readability, the operation code is in **brown**, Mod R/M and SIB bytes are in **red**, and the displacement field is in **black**. Note that this follows the exact format of an instruction: first is the primary opcode, next is the Mod R/M byte, next is the SIB byte, and the displacement byte follows.

If the displacement byte in the above instruction looks odd, please consider that the IA32 and IA64 architectures are **little endian**.

Displacement field

The displacement field is only valid if Mod R/M.mod is mode 1 (Memory+DISP8) or mode 2 (Memory+DISP16 or Memory+DISP32). The displacement can be a byte, word or dword value and is used in conjunction with the Mod R/M and SIB bytes to add displacements to addressing modes. The displacement field always follows the Mod R/M or SIB byte.

Immediate field

The immediate field is only valid if the instruction requires it as an operand. Instructions might require an 8, 16, or 32 bit immediate value. This field must then be present as the last field in the instruction. Instructions that allow both 16 and 32 bit values depend on if an **operand override size** prefix is present to determine the size of this field.

Instruction tables

An **instruction look-up table** is utilized by certain software to help facilitate the machine language translation of instructions. The tables reflect that of a generic instruction table that provides all of the operational codes and operand types for all of the instructions. We can use a resource, such as the IA32 manuals or an online reference, to construct the tables or to help with building machine instructions. The design of these tables varies considerably; it is important to read the documentation on how to read these look-up tables.

The tables would present an instruction in a form similar to the following.

```
| 0x10 | ADC | R/MEM8 | R8 |
+------+-----+--------+----+
```

This represents an ADC instruction whose operation code is 0x10. The R/MEM8 is the first operand, and R8 is the second operand. Operands can be represented in different ways depending on the tables' design. The table may also present additional information such as effected flags the instruction sets, what form of the opcode byte the instruction might use (such as if it stores a register ID in the opcode field), supported processors, and so on. These tables can get really large in size but they all provide the same basic information typically presented in the above form.

R/MEM8 in the above means that the first operand is a "register" or "8 bit memory location". The "R8" means the second operand is an 8 bit register. **If an instruction has a memory operand, then a Mod R/M (and possibly an SIB byte) must follow**. Also, **if an instruction takes two register operands, a Mod R/M byte must follow**. The Mod R/M byte will store the memory addressing mode information or both register codes in Mod R/M.rm and Mod R/M.reg. The CPU will know the register code is for an 8 bit register because of the opcode. **The opcode tells the CPU not only what instruction to execute, but also what operands the instruction requires**. An instruction may utilize different types of operands, because of this the same instruction can occupy multiple opcodes. For example, the above instruction form uses opcode 0x10. Other forms include but are not limited to the following.

```
| 0x11 | ADC | R/MEM16/MEM32 | R16/REG32     |
+------+-----+---------------+---------------+
| 0x12 | ADC | R8            | R/REG16/REG32 |
+------+-----+---------------+---------------+
| 0x13 | ADC | R16/REG32     | R/MEM16/MEM32 |
+------+-----+---------------+---------------+
```

If an instruction uses an operand like **REG16/REG32**, you need to deduce the operand to use based on if an **operand size override** prefix is present and the current CPU operation mode (that is, if it is running in protected mode, real mode, long mode, and so on). For example, if the instruction is an **ADC ax, word ptr [0]** and we are running this in protected mode (or, in assembly language terminology, we used **bits32** or **use32** directives), we can use opcode 0x13 for the instruction. We know this instruction takes the form **ADC REG16, MEM16/MEM32**. AX is the first operand, which is a 16 bit register (REG16). But what is [0]? In order to find out, we take into consideration that there is no address size override and that we are in protected mode. Due to there being no address size override we are to use the native size, which is 32 bit memory addressing. (Please see the section on the address size override prefix for more information.) Using this, we conclude that we select the **ADC REG16, MEM32** form. (If an address size override did exist, we would select the ADC REG16, MEM16 form).

If an instruction only has a single register operand, verify if the operand is stored in the OPCode.reg field. (Please see the Operational code section for more information.) Some instructions do this to save space, **this is what allows single byte instructions**. Some instructions might utilize two registers for operands storing them in OPCode.reg and Mod R/M.reg or Mod R/M.rm.

To complete this example, we note the following: OPCode 0x13, AX register code (0), Addressing mode is [Memory] with a displacement of 0. Due to is being in protected mode, we use the 32 bit Mod R/M form. Mod R/M.reg = 0 (selecting AX), Mod R/M.rm = 5 (DISP32) and Mod R/M.mod = 0 ([MEMORY]). This creates a Mod R/M value of **00 000 101 binary**. Due to us not using a [SIB] mode, we do not need to use an SIB byte. (For an example that does use the SIB, please see our previous example for disassembling **mov eax, [ebx+edi*2+0xa000]**). Using this information, we can build the machine code.

```
0x66 0x13 0x05 0x00 0x00 0x00 0x00
```

The OPCode is in **brown** and Mod R/M byte is in **red**. The displacement byte is a DISP32 (due to Mod R/M.rm) so must be a dword. This is identified in **black**. The 0x66 is an **operand size override prefix** which is in **blue**. We use an operand override size prefix in order to select the REG16 and MEM16 form rather then the REG32 and MEM32 form. If we omit the prefix, we will get the following instead.

```
0x13 0x05 0x00 0x00 0x00 0x00
```

In protected mode, this is an **adc eax, dword ptr [0]** instruction which was not what we wanted. Please see the operand size override prefix section for more information.

If we wanted to turn **ADC ax, word ptr [0]** into **REP ADC ax, word ptr ES:[0]** we can use an ES override prefix and REP prefix:

```
0xf3 0x26 0x66 0x13 0x05 0x00 0x00 0x00 0x00
```

The order of the prefix bytes do not matter.

Resources

The following resources are presented for supplemental reading. Please note that we do not provide support for these resources.

http://ref.x86asm.net/	IA32 and IA64 instruction table
http://www.sandpile.org/	Instruction format tables
http://wiki.osdev.org/X86-64_Instruction_Encoding	IA32 and IA64 Instruction encoding

Conclusion

This chapter provided an overview of machine language programming and the instruction encoding on IA32 and IA64 architectures. A goal of this chapter is to present the material in a new way to encourage the development of debuggers and tool-chains. This chapter can also be used as a reference with an instruction table when emulating certain instructions.

Please let me know if there are any questions or comments,

~Mike ();

OS Development Series Editor

Operating Systems Development Series

Operating Systems Development - Scan Codes

by Mike, 2008

This series is intended to demonstrate and teach operating system development from the ground up.

Introduction

This is a resource that lists all of the scan codes. There are three defined scan code sets for the keybord controller.

Original XT Scan Code Set

Scan Code Set

KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK
A	1E	9E		9	0A	8A		[	1A	9A
B	30	B0		`	29	89		INSERT	E0, 52	E0, D2
C	2E	AE		-	0C	8C		HOME	E0, 47	E0, 97
D	20	A0		=	0D	8D		PG UP	E0, 49	E0, C9
E	12	92		¥	2B	AB		DELETE	E0, 53	E0, D3
F	21	A1		BKSP	0E	8E		END	E0, 4F	E0, CF
G	22	A2		SPACE	39	B9		PG DN	E0, 51	E0, D1
H	23	A3		TAB	0F	8F		U ARROW	E0, 48	E0, C8
I	17	97		CAPS	3A	BA		L ARROW	E0, 4B	E0, CB
J	24	A4		L SHFT	2A	AA		D ARROW	E0, 50	E0, D0
K	25	A5		L CTRL	1D	9D		R ARROW	E0, 4D	E0, CD
L	26	A6		L GUI	E0, 5B	E0, DB		NUM	45	C5
M	32	B2		L ALT	38	B8		KP /	E0, 35	E0, B5
N	31	B1		R SHFT	36	B6		KP *	37	B7
O	18	98		R CTRL	E0, 1D	E0, 9D		KP -	4A	CA
P	19	99		R GUI	E0, 5C	E0, DC		KP +	4E	CE
Q	10	90		R ALT	E0, 38	E0, B8		KP EN	E0, 1C	E0, 9C
R	13	93		APPS	E0, 5D	E0, DD		KP .	53	D3
S	1F	9F		ENTER	1C	9C		KP 0	52	D2
T	14	94		ESC	01	81		KP 1	4F	CF
U	16	96		F1	3B	BB		KP 2	50	D0
V	2F	AF		F2	3C	BC		KP 3	51	D1
W	11	91		F3	3D	BD		KP 4	4B	CB
X	2D	AD		F4	3E	BE		KP 5	4C	CC
Y	15	95		F5	3F	BF		KP 6	4D	CD
Z	2C	AC		F6	40	C0		KP 7	47	C7
0	0B	8B		F7	41	C1		KP 8	48	C8
1	02	82		F8	42	C2		KP 9	49	C9
2	03	83		F9	43	C3		]	1B	9B
3	04	84		F10	44	C4		;	27	A7
4	05	85		F11	57	D7		'	28	A8
5	06	86		F12	58	D8		,	33	B3
6	07	87		PRNT SCRN	E0, 2A, E0, 37	E0, B7, E0, AA		.	34	B4

7	08	88		SCROLL	46	C6		/	35	B5
8	09	89		PAUSE	E1, 1D, 45 E1, 9D, C5	-NONE-				

ACPI Scan Codes

Key	Make Code	Break Code
Power	E0, 5E	E0, DE
Sleep	E0, 5F	E0, DF
Wake	E0, 63	E0, E3

Windows Multimedia Scan Codes

Key	Make Code	Break Code
Next Track	E0, 19	E0, 99
Previous Track	E0, 10	E0, 90
Stop	E0, 24	E0, A4
Play/Pause	E0, 22	E0, A2
Mute	E0, 20	E0, A0
Volume Up	E0, 30	E0, B0
Volume Down	E0, 2E	E0, AE
Media Select	E0, 6D	E0, ED
E-Mail	E0, 6C	E0, EC
Calculator	E0, 21	E0, A1
My Computer	E0, 6B	E0, EB
WWW Search	E0, 65	E0, E5
WWW Home	E0, 32	E0, B2
WWW Back	E0, 6A	E0, EA
WWW Forward	E0, 69	E0, E9
WWW Stop	E0, 68	E0, E8
WWW Refresh	E0, 67	E0, E7
WWW Favorites	E0, 66	E0, E6

Default Scan Code Set for Modern Keyboards

Scan Code Set

KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK
A	1C	F0, 1C		9	46	F0, 46		[	54	F0, 54
B	32	F0, 32		`	0E	F0, 0E		INSERT	E0, 70	E0, F0, 70
C	21	F0, 21		-	4E	F0, 4E		HOME	E0, 6C	E0, F0, 6C
D	23	F0, 23		=	55	F0, 55		PG UP	E0, 7D	E0, F0, 7D
E	24	F0, 24		¥	5D	F0, 5D		DELETE	E0, 71	E0, F0, 71
F	2B	F0, 2B		BKSP	66	F0, 66		END	E0, 69	E0, F0, 69
G	34	F0, 34		SPACE	29	F0, 29		PG DN	E0, 7A	E0, F0, 7A
H	33	F0, 33		TAB	0D	F0, 0D		U ARROW	E0, 75	E0, F0, 75
I	43	F0, 43		CAPS	58	F0, 58		L ARROW	E0, 6B	E0, F0, 6B
J	3B	F0, 3B		L SHFT	12	F0, 12		D ARROW	E0, 72	E0, F0, 72
K	42	F0, 42		L CTRL	14	F0, 14		R ARROW	E0, 74	E0, F0, 74
L	4B	F0, 4B		L GUI	E0, 1F	E0, F0, 1F		NUM	77	F0, 77

M	3A	F0, 3A		L ALT	11	F0, 11		KP /	E0, 4A	E0, F0, 4A
N	31	F0, 31		R SHFT	59	F0, 59		KP *	7C	F0, 7C
O	44	F0, 44		R CTRL	E0, 14	E0, F0, 14		KP -	7B	F0, 7B
P	4D	F0, 4D		R GUI	E0, 27	E0, F0, 27		KP +	79	F0, 79
Q	15	F0, 15		R ALT	E0, 11	E0, F0, 11		KP EN	E0, 5A	E0, F0, 5A
R	2D	F0, 2D		APPS	E0, 2F	E0, F0, 2F		KP .	71	F0, 71
S	1B	F0, 1B		ENTER	5A	F0, 5A		KP 0	70	F0, 70
T	2C	F0, 2C		ESC	76	F0, 76		KP 1	69	F0, 69
U	3C	F0, 3C		F1	05	F0, 05		KP 2	72	F0, 72
V	2A	F0, 2A		F2	06	F0, 06		KP 3	7A	F0, 7A
W	1D	F0, 1D		F3	04	F0, 04		KP 4	6B	F0, 6B
X	22	F0, 22		F4	0C	F0, 0C		KP 5	73	F0, 73
Y	35	F0, 35		F5	03	F0, 03		KP 6	74	F0, 74
Z	1A	F0, 1A		F6	0B	F0, 0B		KP 7	6C	F0, 6C
0	45	F0, 45		F7	83	F0, 83		KP 8	75	F0, 75
1	16	F0, 16		F8	0A	F0, 0A		KP 9	7D	F0, 7D
2	1E	F0, 1E		F9	01	F0, 01		]	5B	F0, 5B
3	26	F0, 26		F10	09	F0, 09		;	4C	F0, 4C
4	25	F0, 25		F11	78	F0, 78		'	52	F0, 52
5	2E	F0, 2E		F12	07	F0, 07		,	41	F0, 41
6	36	F0, 36		PRNT SCRN	E0, 12, E0, 7C	E0, F0, 7C, E0, F0, 12		.	49	F0, 49
7	3D	F0, 3D		SCROLL	7E	F0, 7E		/	4A	F0, 4A
8	3E	F0, 3E		PAUSE	E1, 14, 77, E1, F0, 14, F0, 77	-NONE-				

ACPI Scan Codes

Key	Make Code	Break Code
Power	E0, 37	E0, F0, 37
Sleep	E0, 3F	E0, F0, 3F
Wake	E0, 5E	E0, F0, 5E

Windows Multimedia Scan Codes

Key	Make Code	Break Code
Next Track	E0, 4D	E0, F0, 4D
Previous Track	E0, 15	E0, F0, 15
Stop	E0, 3B	E0, F0, 3B
Play/Pause	E0, 34	E0, F0, 34
Mute	E0, 23	E0, F0, 23
Volume Up	E0, 32	E0, F0, 32
Volume Down	E0, 21	E0, F0, 21
Media Select	E0, 50	E0, F0, 50
E-Mail	E0, 48	E0, F0, 48
Calculator	E0, 2B	E0, F0, 2B
My Computer	E0, 40	E0, F0, 40
WWW Search	E0, 10	E0, F0, 10
WWW Home	E0, 3A	E0, F0, 3A
WWW Back	E0, 38	E0, F0, 38

WWW Forward	E0, 30	E0, F0, 30
WWW Stop	E0, 28	E0, F0, 28
WWW Refresh	E0, 20	E0, F0, 20
WWW Favorites	E0, 18	E0, F0, 18

PS/2 Scan Code Set for AT Motherboards

KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK	-----	KEY	MAKE	BREAK
A	1C	F0, 1C		9	46	F0, 46		[	54	F0, 54
B	32	F0, 32		`	0E	F0, 0E		INSERT	67	F0, 67
C	21	F0, 21		-	4E	F0, 4E		HOME	6E	F0, 6E
D	23	F0, 23		=	55	F0, 55		PG UP	6F	F0, 6F
E	24	F0, 24		¥	5C	F0, 5C		DELETE	64	F0, 64
F	2B	F0, 2B		BKSP	66	F0, 66		END	65	F0, 65
G	34	F0, 34		SPACE	29	F0, 29		PG DN	6D	F0, 6D
H	33	F0, 33		TAB	0D	F0, 0D		U ARROW	63	F0, 63
I	43	F0, 48		CAPS	14	F0, 14		L ARROW	61	F0, 61
J	3B	F0, 3B		L SHFT	12	F0, 12		D ARROW	60	F0, 60
K	42	F0, 42		L CTRL	11	F0, 11		R ARROW	6A	F0, 6A
L	4B	F0, 4B		L WIN	8B	F0, 8B		NUM	76	F0, 76
M	3A	F0, 3A		L ALT	19	F0, 19		KP /	4A	F0, 4A
N	31	F0, 31		R SHFT	59	F0, 59		KP *	7E	F0, 7E
O	44	F0, 44		R CTRL	58	F0, 58		KP -	4E	F0, 4E
P	4D	F0, 4D		R WIN	8C	F0, 8C		KP +	7C	F0, 7C
Q	15	F0, 15		R ALT	39	F0, 39		KP EN	79	F0, 79
R	2D	F0, 2D		APPS	8D	F0, 8D		KP .	71	F0, 71
S	1B	F0, 1B		ENTER	5A	F0, 5A		KP 0	70	F0, 70
T	2C	F0, 2C		ESC	08	F0, 08		KP 1	69	F0, 69
U	3C	F0, 3C		F1	07	F0, 07		KP 2	72	F0, 72
V	2A	F0, 2A		F2	0F	F0, 0F		KP 3	7A	F0, 7A
W	1D	F0, 1D		F3	17	F0, 17		KP 4	6B	F0, 6B
X	22	F0, 22		F4	1F	F0, 1F		KP 5	73	F0, 73
Y	35	F0, 35		F5	27	F0, 27		KP 6	74	F0, 74
Z	1A	F0, 1A		F6	2F	F0, 2F		KP 7	6C	F0, 6C
0	45	F0, 45		F7	37	F0, 37		KP 8	75	F0, 75
1	16	F0, 16		F8	3F	F0, 3F		KP 9	7D	F0, 7D
2	1E	F0, 1E		F9	47	F0, 47		]	5B	F0, 5B
3	26	F0, 26		F10	4F	F0, 4F		;	4C	F0, 4C
4	25	F0, 25		F11	56	F0, 56		'	52	F0, 52
5	2E	F0, 2E		F12	5E	F0, 5E		,	41	F0, 41
6	36	F0, 36		PRNT SCRN	57	F0, 57		.	49	F0, 49
	3D	F0, 3D		SCROLL	5F	F0, 5F		/	4A	F0, 4A
8	3E	F0, 3E		PAUSE	62	F0, 62				

Conclusion

Until next time,

~Mike ();
BrokenThorn Entertainment. Currently developing EvolutionEngine and MicroOS Operating System.

Questions or comments? Feel free to Contact me.