Operating Systems Development Series

Operating Systems Development - PIC, PIT, and exceptions

by Mike, 2008

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

Please note: This tutorial covers hardware interrupt handling, not software interrupt handling. If you are looking for software interrupts, please see Tutorial 15. This tutorial requires knowledge of software interrupt handling.

Introduction

Welcome to...what? Tutorial 16 already?

In the last tutorial, we have dived deep into the world of interrupt handling. We have covered, and even implimented, interfaces for the GDT and IDT inside of our **hardware abstraction layer**. We have covered almost everything we needed for software interrupt handling to work. ...But WAIT! What about hardware interrupts? o_0

Because alot of critical system devices use interrupts, it is necessary for us to be able to handle and catch interrupts triggered by hardware devices. The good news? This is already done for us! By what? The **8259 Programmable Interrupt Controller (PIC)**. We will look closer in the next section.

Even if we get hardware interrupts working by itself, we will still be running into problems do to the system timer. As long as the system timer does not use a valid interrupt handler that we have set up for it, it will triple fault a few milliseconds after you enable hardware interrupts. After all, it will call an invalid interrupt handler, remember? Thus, we will also fix this small problem by reprogramming the system timer, otherwise known as the **Programmable Interval Timer (PIT)**.

Alot of stuff in this tutorial. We will look at:

- Hardware Interrupts
- Interrupt Chaining
- Hal: Programmable Interrupt Controller
- Hal: Programmable Interval Timer
- Hardware Abstraction
- Interrupts Implimentation and Design for our HAL

Please note that this does not cover interrupt handling. Please see Tutorial 15 for interrupt handlng.

With all of that in mind, lets take a look...

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 on the keyboard, or a single clock tick on the internal timer, for example.

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

Lets take a look at these hardware interrupts.

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

You do not need to worry to much about each device just yet. The 8259A Pins are described in detail in the 8259 PIC tutorial. The Interrupt Numbers listed in this 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.

Wait...What is this PIC thing? All of these hardware devices that can signal hardware devices are connected indirectly to the **8259A Programmable Interrupt Controller (PIC)**. This is a special, and very important microcontroller that is used to signal the microprocessor when it needs to fire a hardware interrupt.

We will be programming this microcontroller a little later in this tutorial. Because this microcontroller is fairly complex, we have dedicated another tutorial for it. Please read it here.

Interrupt Chaining

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

But wait... How does the PIC know what IRQs to call within our IDT? *We tell it*.

This is why we must reprogram the PIC in order to let it know what interrupts to use.

Okay, lets say we now have interrupt handlers to handle software and hardware interrupts. What now? How does this work from our perspecitive? Sure, we can easily install handlers for different devices, but what if multiple devices require the same interrupt? What about multiple functions for a software interrupt? This is where **Interrupt Chaining** comes in.

Interrupt chaning is a technique used to restore and call all of the interrupt handlers that share that same interrupt number. This is done by saving the previus **Interrupt Routine (IR)** in a function pointer. Then, installing the new handler, and calling the previus interrupt handler whenever the new IR is called.

Here is an example:

```
void deviceInitialize () {

	//store previus interrupt handler
	prevhandler = getvect (0);

	//install new interrupt handler
	setvect (0, handler);
}

void deviceShutdown () {

	//install previus interrupt handler
	setvect (0, prevhandler);
}
```

```
void handler () {

	// do stuff...

	// call previus interrupt handler
	(*prevhandler) ();
}
```

As you can see, interrupt chaining is rather easy. Notice how the previus interrupt handlers will always be called whenever this interrupt fires? **setvect()** installs a new interrupt vector. **getvect()** returns an interrupt vector. These interrupt vectors can be stored in either the **Interrupt Vector Table (IVT)** or **Interrupt Descriptor Table (IDT)**. Wait, what? Thats right--ours :)

Get Ready - Implimenting Interrupt Handling

We have covered alot of ground with interrupts and interrupt handling. Text alone can only go so far. We have even looked a little bit about how hardware interrupt handling works, but not enough to get very far.

We cannot impliment hardware interrupt handling until we learn how to program the **Programmable Interrupt Controller**. Also, we cannot enable hardware interrupts until we fix the timing problem (Remember that the **Programmable Interval Timer** is still connected to IRQ8 thanks to the BIOS? That means, as soon as we re-enable hadware interrupts, the next timer tick will result in a double fault.) Because of this, we also have to learn how to reprogram the **Programmable Interval Timer**.

This, dear readers, is where things get complex. **Welcome back to the world of hardware programming** :)

There is good news, however... None of these microcontrollers are very complex. However, to keep the main series from getting too complex, I have decided to write **2 tutorials dedicated to these microcontrollers.** This is required reading for understanding of the demo and code that lies ahead.

Because of this, I recommend for all of our readers to read the following tutorials before continuing:

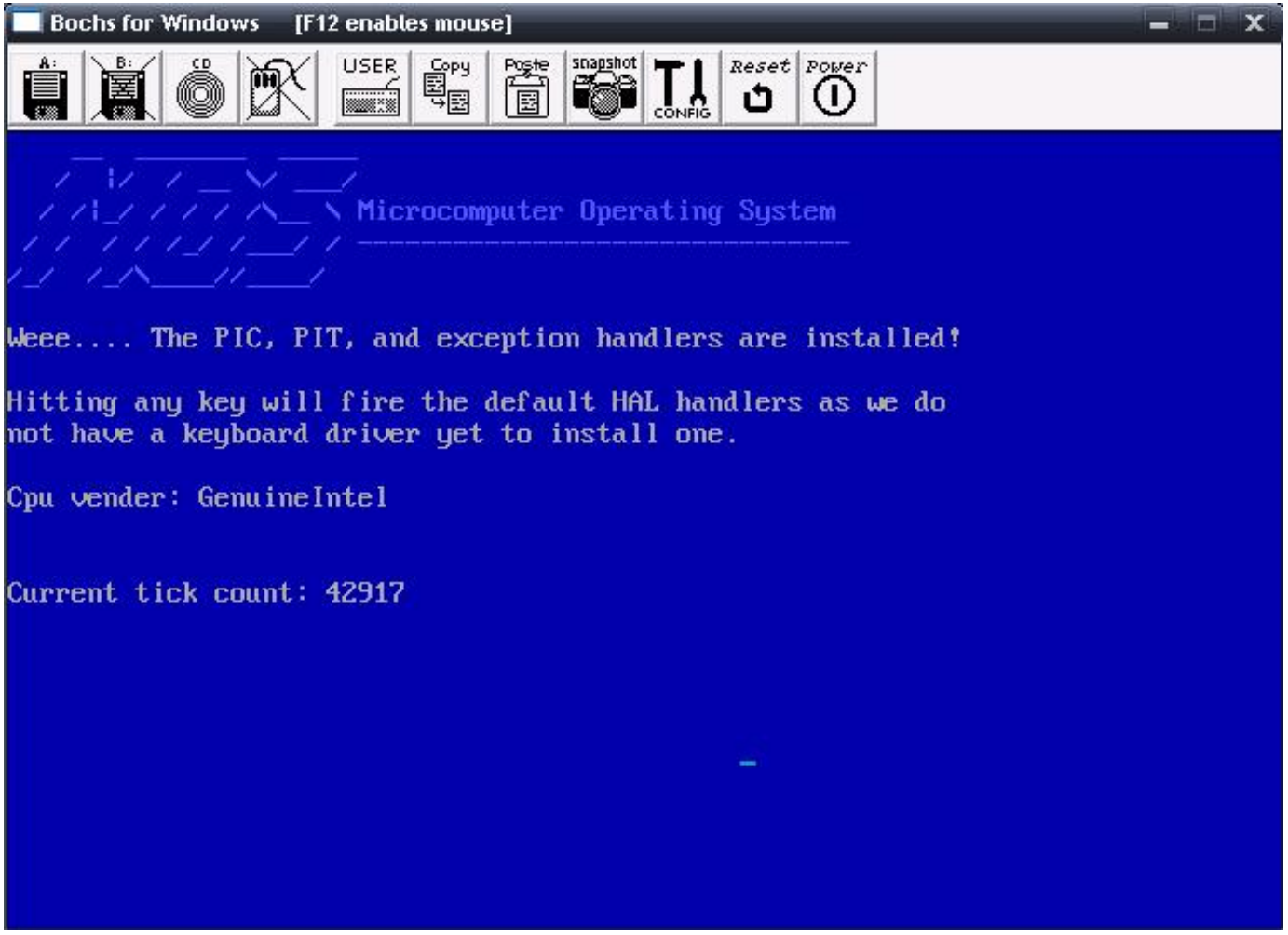
- 8259 Programmable Interrupt Controller
- 8253 Programmable Interval Timer

Do not worry if you do not understand everything in these tutorials yet. You will see us implimenting everything within the above tutorials in the sections to come. :) I will also still be describing everything here, so I will not lead you in the dark.

It may also be helpful to use the above tutorials as a refrence, as we will be refrencing the above tutorials alot throughout the upcoming sections.

So..? What are you waiting for? Jump into those tutorials! And come back here when you are done. Don't worry, I am just text... I will still be here when you get back.. ...unless I am deleted of course :)

Demo



The new demo in action

Demo Download (MSVC++)

I am going to admit, this demo is a little complex. It is not too hard, though, as most of the code should look familiar.

This demo uses our **hardware abstraction layer (HAL)** to install its own exception handlers. This allows us to catch processor exception errors (Like divide by 0.) It also initializes the HAL, which in turns initializes processor tables, the PIC, and the PIT. This allows us to enable hardware exceptions (Finally!) and start a timer. **The demo displays the current tick count (Updated by the PIT's interrupt handler) on screen.** Any other interrupt generated will be handled by the IDT's default interrupt handler. We have set this up in our IDT tutorial remember?

Because the keyboard generates IRQ 9, and we have not defined an interrupt for it yet, anytime a key is pressed the default handler for the IDT will execute.

Okay... I have tried to make this demo look a little cooler then the other ones. At the same time, I did not want to add even more complexity to it. Because we have looked at creating interfaces for the PIC and PIT microcontrollers, lets look a little bit at the demo code itself, shall we..?

Hardware Abstraction

The first interface that we will look at is the one provided by the hardware abstraction layer. This can be seen by looking at **include/hal.h** and **hal/hal.cpp**. I will not be describing the routines in depth as most of it is very simple and simply use the other interfaces (GDT, IDT, CPU, PIC, PIT, etc) that we have developed (And are about to develop). Instead, I want to look at the interface itself. This will be the interface used by the kernel and device drivers, so why not?

The new hal.h

This is where we start seeing how useful hardware abstraction can be. I wanted to provide a "DOS"-like interface that is just as easy to use as programming 16bit DOS is. In doing so, I came up with a very easy list of routines that can be used for alot of different purposes. Looking at these routines, you will see there is absolutley no refrence to the hardware devices or tables that are used by them. This is what hardware abstraction is all about. It does not abstract the architecture; but rather the hardware that it uses.

Alot of the code that we use later use the routines within the HAL to perform its task. Because of this, I wanted you to take a look at the hardware abstraction layer now, and the routines it provides.

```
extern     int               _cdecl     hal_initialize ();
extern     int               _cdecl     hal_shutdown ();
extern     void              _cdecl     enable ();
extern     void              _cdecl     disable ();
extern     void              _cdecl     geninterrupt (int n);
extern     unsigned char          _cdecl     inportb (unsigned short id);
extern     void              _cdecl     outportb (unsigned short id, unsigned char value);
extern     void              _cdecl     setvect (int intno, void (_cdecl far &vect) ( ) );
extern     void (_cdecl      far *   _cdecl    getvect (int intno)) ( );
extern     bool              _cdecl     interruptmask (uint8_t intno, bool enable);
extern     inline void            _cdecl     interruptdone (unsigned int intno);
extern     void              _cdecl     sound (unsigned frequency);
extern  const char*          _cdecl     get_cpu_vender ();
extern     int               _cdecl     get_tick_count ();
```

If you have ever programmed 16bit DOS, you should feel at home right about now! :)

Programmable Interrupt Controller

8259: Microcontroller

The 8259 Microcontroller familiy is a set of **Programmable Interrupt Controller (PIC) Integrated Circuits (ICs)**. Hardware controllers are indirectly connected to the PIC when a hardware interrupt is requested. Because of this, in order to handle hardware interrupts, we must have an understanding of how to program this microcontroller.

I will still go over everything here, however the 8259 is a complex microcontroller. Because of this, we have dedicated a full tutorial to cover just this controller. Because of this, **in order to get the most out of this section, Please see (and refrence) the following tutorial to learn about the PIC:**

8259A Programmable Interrupt Controller Tutorial

Please note: **We will not cover everything about the PIC nor hardware interrupt handling here. Please see the above tutorial for this.**

8259: Abstract

The **Programmable Interrupt Controller (PIC)** is a microcontroller used to provide the connection between devices and the processor through **interrupt lines**. This allows devices to signal the processor whenever it requires attention from the system software or executive. This is the **Interrupt Request (IRQ)**.

The PIC controls all of the hardware interrupt requests. It allows us to recieve signals from different hardware devices whenever they require attention. When a device, such as the **Floppy Disk Controller (FDC)** requires attention, it tells the PIC to fire the IRQ it is assigned to. From here, the PIC will signal the processor, and give the interrupt number to call. The processor then offsets into the IDT, and executes the interrupt handler at ring 0. We define all of the interrupt handlers, so we now take control.

The best thing about this is that it is all **automatic** thanks to the PIC. Whenever a device signals the PIC, our interrupt handler will be executed automatically. The processor also performs a **task switch** to ring 0, so we will always end up in kernal land to handle the request. Cool, huh?

The PIC itself is a complex microcontroller. I will try to cover everything in detail here, but please keep in mind that--in order to get the most out of this tutorial, we encourage our readers to read the above PIC tutorial.

With all of that in mind--lets dive into the interface. All of this code can be found in the demo at the end of this tutorial.

Operation Commands

An **Operation Command** is a special command that is composed of a bit pattern. This bit pattern must be set up to describe the command for a microcontroller. There are basically two types of operation commands: **Initialization Command Words (ICWs)** and **Operation Command Words (OCWs)**.

ICWs are operation commands that must only be used during the initialization of the device. OCWs are used to control the device after the device has been initialized.

pic.h: Interface

This file provides the overall minidriver interface for the rest of the system. This is the interface to controlling and managing the PIC. I define a "minidriver" as a driver embedded in a peice of software, and not as stand alone software.

pic.h: Device Connections

In the PIC tutorial, we have looked at hardware interrupts in alot of depth. We have looked at how hardware devices signal the PIC whenever it requires attention of the system software or executive. For this to work, each device is indirectly connected to an **Interrupt Request (IR)** line on the PIC. This line not only represents the **Interrupt Request (IRQ)** the device uses but also its pririty level (The lower the IRQ number, the higher pririty.)

To help when working with individual devices and their IRQs, we will want to abstract the IRQ they use. This helps increase portability but also readability as they are behind nice constants. Remember: Magic numbers are bad!

```
//! The following devices use PIC 1 to generate interrupts
#define		I86_PIC_IRQ_TIMER		0
#define		I86_PIC_IRQ_KEYBOARD		1
#define		I86_PIC_IRQ_SERIAL2		3
#define		I86_PIC_IRQ_SERIAL1		4
#define		I86_PIC_IRQ_PARALLEL2		5
#define		I86_PIC_IRQ_DISKETTE		6
#define		I86_PIC_IRQ_PARALLEL1		7

//! The following devices use PIC 2 to generate interrupts
#define		I86_PIC_IRQ_CMOSTIMER		0
#define		I86_PIC_IRQ_CGARETRACE		1
#define		I86_PIC_IRQ_AUXILIARY		4
#define		I86_PIC_IRQ_FPU			5
#define		I86_PIC_IRQ_HDC			6
```

The above constants list all of the devices (along with their IRQ line/number) that they use. **There are only 8 IR lines per PIC**, hence only 8 possible IRQs per PIC. **Remember that PIC's can be cascaded with secondary PICs** (Up to 8 PICs can be cascaded with each other.) Typical x86 architectures only have 2--One primary and one secondary.

The two most important devices for us right now are the timer (I86_PIC_IRQ_TIMER) and keyboard (I86_PIC_IRQ_KEYBOARD). We will be using I86_PIC_IRQ_TIMER in this tutorial, so you will see how everything works together, cool?

pic: 8259 Commands

Setting up the PIC is farily complex. It is done through a series of **Command Words**, 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. We will first look at the **Operation Command Word (OCW)** that are used to control the PIC. We will look at the initialization commands a little later.

pic: Operation Command Word 1

This represents the value in the **Interrupt Mask Register (IMR)**. It does not have a special format, so it is handled directly in the implimentation

file to enable and disable hardware interrupts. It is a single byte in size. We enable and disable ("mask and unmask") an interrupt request line by setting the correct bit. Remember that there are only 8 IRQ's per PIC? So, bit 0 in the IMR is IRQ 0, bit 1 is IRQ 1, bit 2 is IRQ 2, and so on.

We will take a look at the **Interrupt Mask register** a little later on, cool?

pic: Operation Command Word 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!

The format of OCW 2 is very easy. The first three bits are the current interrupt level. Bits 3-4 are reserved (Must be 0). Bit 5 represents **End of Interrupt (EOI)**. Bit 6 is the **Selection** bit. Bit 7 provides a **Rotation** command.

Because each command is selected in individual bits, we can **bitwise OR** these commands together to produce OCW 2.

```
//! Command Word 2 bit masks. Use when sending commands
#define		I86_PIC_OCW2_MASK_L1		1		//00000001	//Level 1 interrupt level
#define		I86_PIC_OCW2_MASK_L2		2		//00000010	//Level 2 interrupt level
#define		I86_PIC_OCW2_MASK_L3		4		//00000100	//Level 3 interrupt level
#define		I86_PIC_OCW2_MASK_EOI		0x20	//00100000	//End of Interrupt command
#define		I86_PIC_OCW2_MASK_SL		0x40	//01000000	//Select command
#define		I86_PIC_OCW2_MASK_ROTATE	0x80	//10000000	//Rotation command
```

There you have it! This is an important command word for us. We will be required to send this command word from all interrupt handlers.

Remember that the PIC masks off the interrupt when it gets executed? This means that no more interrupt requests on that IR line can execute until the processor acknowledges the PIC. This is done by sending an **End of Interrupt** command word to the correct PIC. We can do this by **masking off** the EOI bit in the command word. This is what **I86_PIC_OCW2_MASK_EOI** is used for.

A little later on, you will see that the interface has a **i86_pic_send_command** routine that is used to...erm...send commands to the PIC. Lets look at an example of sending the EOI command using this routine so that you can see how it works:

```
i86_pic_send_command (I86_PIC_OCW2_MASK_EOI, picNumber);
```

The above code will send an EOI command to the pic in **picNumber**, cool?

I suppose thats it for OCW 2. On to the next one!

pic: Operation Command Word 3

I plan on adding to this section.

```
//! Command Word 3 bit masks. Use when sending commands
#define		I86_PIC_OCW3_MASK_RIS		1		//00000001
#define		I86_PIC_OCW3_MASK_RIR		2		//00000010
#define		I86_PIC_OCW3_MASK_MODE		4		//00000100
#define		I86_PIC_OCW3_MASK_SMM		0x20		//00100000
#define		I86_PIC_OCW3_MASK_ESMM		0x40		//01000000
#define		I86_PIC_OCW3_MASK_D7	0x80	//10000000
```

pic.cpp: Implimentation

Okay...Everything was easy so far, right? You are probably asking "Where is the challenge!??" Well, okay then.

pic.cpp provides the implimentation for our PIC interface. First thing we must look at are the registers.

pic.cpp: Register constants

This is where we define the constants to abstract the port locations for the PICs. Notice that I have defined constants for all register names, even though they share the same port address. The reason is for completness: Even though they share the same port location, they still are different registers.

```
//! PIC 1 register port addresses
#define I86_PIC1_REG_COMMAND		0x20		// command register
#define I86_PIC1_REG_STATUS		0x20		// status register
#define I86_PIC1_REG_DATA		0x21		// data register
#define I86_PIC1_REG_IMR		0x21		// interrupt mask register (imr)

//! PIC 2 register port addresses
#define I86_PIC2_REG_COMMAND		0xA0			// ^ see above register names
#define I86_PIC2_REG_STATUS		0xA0
#define I86_PIC2_REG_DATA		0xA1
```

```
#define I86_PIC2_REG_IMR		0xA1
```

Not to hard. We send commands to the **command register**, and read data from the **data register**. If we are writing from a data register, we are accessing the **Interrupt Mask Register (IMR)** which can be used to manually mask off or unmask interrupt requests. This is how we enable or disable interrupt requests.

The register we are accessing depends on wether it is a write or read operation. If we **write** to port 0x20, we are accessing the command register. If we are **reading** from it, we are accessing the status register.

Lastly, because this is an implimentation detail, it is part of the implimentation (pic.cpp), not interface.

Lets take a look at the constants used during initialization next.

pic.cpp: Initialization Control Word 1

This is the primary control word used when initializing the PICs. 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.

There are two types of constants for each command word. The first type are **bit masks** that are used to mask off the bits that the data represents. The second type of constants used are **command control bits** which are used in conjunction with the masks to set them to their correct values.

Lets look closer. Here are the ICW 1 bit masks. Notice how they follow the format shown in the above table. We do not define anything for the last three bits as they are always zero for x86 architectures.

```
//! Initialization Control Word 1 bit masks
```

```
#define I86_PIC_ICW1_MASK_IC4           0x1  //00000001     // Expect ICW 4 bit
#define I86_PIC_ICW1_MASK_SNGL          0x2  //00000010     // Single or Cascaded
#define I86_PIC_ICW1_MASK_ADI           0x4  //00000100     // Call Address Interval
#define I86_PIC_ICW1_MASK_LTIM          0x8  //00001000     // Operation Mode
#define I86_PIC_ICW1_MASK_INIT          0x10//00010000     // Initialization Command
```

Okay...We can easily use the above bit masks to set the bits in the ICW 1. But, how do we know what they mean? That is, when we mask off the bits that we are wanting to set, how do we know what the value we are setting them to mean? This is where **command control bits** come in.

They contain the constant values that may be used to set the above masked off bits to. This helps increase readability and extendability alot.

Here are the command control bits for ICW 1. Lets take a look...

```
#define I86_PIC_ICW1_IC4_EXPECT         1     //1         //Use when setting I86_PIC_ICW1_MASK_IC4
#define I86_PIC_ICW1_IC4_NO         0     //0
#define I86_PIC_ICW1_SNGL_YES           2     //10        //Use when setting I86_PIC_ICW1_MASK_SNGL
#define I86_PIC_ICW1_SNGL_NO            0     //00
#define I86_PIC_ICW1_ADI_CALLINTERVAL4      4     //100           //Use when setting I86_PIC_ICW1_MASK_ADI
#define I86_PIC_ICW1_ADI_CALLINTERVAL8      0     //000
#define I86_PIC_ICW1_LTIM_LEVELTRIGGERED    8     //1000          //Use when setting I86_PIC_ICW1_MASK_LTIM
#define I86_PIC_ICW1_LTIM_EDGETRIGGERED         0     //0000
#define I86_PIC_ICW1_INIT_YES           0x10//10000        //Use when setting I86_PIC_ICW1_MASK_INIT
#define I86_PIC_ICW1_INIT_NO            0     //00000
```

Not to hard. The naming convention used allows us to easily know what to use, and where. For example, **I86_PIC_ICW1_SNGL_YES** is used with **I86_PIC_ICW1_MASK_SNGL**, **I86_PIC_ICW1_LTIM_EDGETRIGGERED** is used with **I86_PIC_ICW1_MASK_LTIM**.

Here is an example of how they work together. When we are initializing the PIC, we will need to enable initialization, and to send ICW 4. To do this, we simply set up ICW 1 like this:

```
uint8_t     icw=0;
icw = (icw & ~I86_PIC_ICW1_MASK_INIT) | I86_PIC_ICW1_INIT_YES;
icw = (icw & ~I86_PIC_ICW1_MASK_IC4) | I86_PIC_ICW1_IC4_EXPECT;
```

Thats it!? Yep. Notice how everything works and fits together. This is used throughout the implimentations to set specific bits (or a series of bits) to known values. The best thing here as that--just by looking at the above code--you know what it is doing. (Begin initialization, and to expect ICW 4). Pretty cool, huh? We will be using this method throughout this series when needed when setting and masking off bits.

Initialization Control Word 2

This control word is used to map the base address of the IVT of which the PIC are to use.

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.

Because this command does not have a complex format, it is handled directly inside of **pic.cpp** and does not have any constants.

Initialization Control Word 3

This command word is used to let the PIC controllers know how they are cascaded. To cascade multiple PICs, we must connect one of the PIC's IR lines to each other. We use this command word to let them know what line it is.

Initialization Control Word (ICW) 3		
Bit Number	Value	Description
0-7	S0-S7	Specifies what Interrupt Request (IRQ) is connected to slave PIC

Because this command does not have a complex format, it is handled directly inside of **pic.cpp** and does not have any constants.

Initialization Control Word 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 complex command word, but not to bad. Lets take a look at our defined bit masks. Notice how they follow the format shown above.

```
//! Initialization Control Word 4 bit masks
#define I86_PIC_ICW4_MASK_UPM        0x1  //00000001     // Mode
#define I86_PIC_ICW4_MASK_AEOI       0x2  //00000010     // Automatic EOI
#define I86_PIC_ICW4_MASK_MS         0x4  //00000100     // Selects buffer type
#define I86_PIC_ICW4_MASK_BUF        0x8  //00001000     // Buffered mode
#define I86_PIC_ICW4_MASK_SFNM       0x10//00010000     // Special fully-nested mode
```

Simular to **ICW 1**, we have a set of control bits that are used in conjunction with the bit masks to set properties. Here they are...

```
#define I86_PIC_ICW4_UPM_86MODE          1     //1          //Use when setting I86_PIC_ICW4_MASK_UPM
#define I86_PIC_ICW4_UPM_MCSMODE  0     //0
#define I86_PIC_ICW4_AEOI_AUTOEOI    2     //10        //Use when setting I86_PIC_ICW4_MASK_AEOI
#define I86_PIC_ICW4_AEOI_NOAUTOEOI      0     //00
#define I86_PIC_ICW4_MS_BUFFERMASTER   4     //100             //Use when setting I86_PIC_ICW4_MASK_MS
#define I86_PIC_ICW4_MS_BUFFERSLAVE      0     //000
#define I86_PIC_ICW4_BUF_MODEYES    8     //1000          //Use when setting I86_PIC_ICW4_MASK_BUF
#define I86_PIC_ICW4_BUF_MODENO          0     //0000
#define I86_PIC_ICW4_SFNM_NESTEDMODE  0x10//10000         //Use when setting I86_PIC_ICW4_MASK_SFNM
#define I86_PIC_ICW4_SFNM_NOTNESTED      0     //00000
```

This is simple snough, huh? ^_^ We can use the above control bits in conjunction with the bit masks to build up the control word. The naming convention used allows us to easily identify what bit masks they are used with.

I suppose thats it for the constants used in the implimentation. Lets get on with the functions...

i86_pic_send_command (): Sends a command to a PIC

This routine sends a command byte to the PIC's command register. **picNum** is a zero-based index representing the PIC we are accessing. On x86, this should either be a 0 or 1. Notice that we test what PIC we are working with in order to get the correct command register.

While this is part of the interface, it should not be used that much outside of the interface. It provides a method so we can manually send and control the PICs, if needed. **This will be required by the interrupt handlers to send the EOI command.**

```
inline void i86_pic_send_command (uint8_t cmd, uint8_t picNum) {

	if (picNum > 1)
		return;
```

```
	uint8_t	reg = (picNum==1) ? I86_PIC2_REG_COMMAND : I86_PIC1_REG_COMMAND;
	outportb (reg, cmd);
}
```

i86_pic_send_data () and i86_pic_read_data (): Send and return a data byte to or from a PIC

These routine are very simular to the above routine, however it writes or reads to the PIC's data register depending on the PIC in **picNum**. Notice how both of these routines are **inline**. Because these routines are small, we want to take out the function call.

```
inline void i86_pic_send_data (uint8_t data, uint8_t picNum) {

	if (picNum > 1)
		return;

	uint8_t	reg = (picNum==1) ? I86_PIC2_REG_DATA : I86_PIC1_REG_DATA;
	outportb (reg, data);
}

inline uint8_t i86_pic_read_data (uint8_t picNum) {

	if (picNum > 1)
		return 0;

	uint8_t	reg = (picNum==1) ? I86_PIC2_REG_DATA : I86_PIC1_REG_DATA;
	return inportb (reg);
}
```

i86_pic_initialize (): Initializes the PICs

This is the final routine for the PIC interface. This initializes both PICs for operation using all of the routines above, and our constants defined for the initialization control words.

This routine is not too complex. Or, rather, not as complex as it looks ;) All it does is send the initialization command to the PIC. It does this by setting the **I86_PIC_ICW1_INIT_YES** bit in the command word. We also set the **I86_PIC_ICW1_IC4_EXPECT** bit. This insures that the controller expects us to send ICW 4. Notice how the constants help improve readability?

The ICW is stored in..well... **icw**. We send the command to both PICs using our **i86_pic_send_command()** routine.

After ICW 1 is sent, we begin initialization by sending ICW 2. Remember that ICW 2 containes the base interrupt numbers? This is passed into the **base0** and **base1** parameters.

Afterwords, we simply send ICW 3. Remember that ICW 3 provides the connection between the master and secondary PIC controllers.

Lastly is ICW 4. We set up x86 mode by setting the **I86_PIC_ICW4_UPM_86MODE** bit. Compare this routine with the example found in the PIC tutorial and be amazed...very amazed on their simularities!

```
//! Initialize pic
void i86_pic_initialize (uint8_t base0, uint8_t base1) {

	uint8_t		icw   = 0;

	//! Begin initialization of PIC

	icw = (icw & ~I86_PIC_ICW1_MASK_INIT) | I86_PIC_ICW1_INIT_YES;
	icw = (icw & ~I86_PIC_ICW1_MASK_IC4) | I86_PIC_ICW1_IC4_EXPECT;

	i86_pic_send_command (icw, 0);
	i86_pic_send_command (icw, 1);

	//! Send initialization control word 2. This is the base addresses of the irq's

	i86_pic_send_data (base0, 0);
	i86_pic_send_data (base1, 1);

	//! Send initialization control word 3. This is the connection between master and slave.
	//! ICW3 for master PIC is the IR that connects to secondary pic in binary format
	//! ICW3 for secondary PIC is the IR that connects to master pic in decimal format

	i86_pic_send_data (0x04, 0);
	i86_pic_send_data (0x02, 1);

	//! Send Initialization control word 4. Enables i86 mode

	icw = (icw & ~I86_PIC_ICW4_MASK_UPM) | I86_PIC_ICW4_UPM_86MODE;

	i86_pic_send_data (icw, 0);
	i86_pic_send_data (icw, 1);
}
```

Whew, I guess thats all of the big stuff for the PIC. All that is left is reprogramming the PIT. Don't worry--its not as complex as the PIC is. Lets take a look...

Programmable Interval Timer

Okay... So the PIC is ready to go, so we can now enable hardware interrupts, right? Yep--Kind of. While everything is okay so far, we still do not have

an interrupt handler installed for the PIT yet. So, what will happen on the next timer tick? ...I think you know where I am getting at :)

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.

On x86 architectures, **The PIT acts as the system timer**, and is **connected to the PIC's IR0 line.** This allows the PIT to fire **IRQ 0** each timer tick. Because of this, we will need to reprogram this microcontroller before we can use it.

The PIT is a complex microcontroller to program. Because of this, we have created a separate tutorial for it. While I will still try to cover everything in detail, **I will not cover everything about the PIT here.**

Please see (and refrence) the following tutorial to learn about the PIT:

8253 Programmable Interval Timer Tutorial

pit.h: Interface

The good thing about the PIT is that it is not that complex to program. It does not contain that much commands, and yet does not need that much commands. It is a small, but powerful chip used for hardware timing and requests.

Operation Command Word

The PIT only containes one **Operation Command Word (OCW)** which is used to initialize a counter. It sets up the counters counting mode, operation mode, and allows us to set up an initil count value.

The command word is a little complex. Here is the complete command word:

- **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

Simular to the PIC's interface, we set up several bit masks that are used to describe the format of the command. Here it is...

```
#define         I86_PIT_OCW_MASK_BINCOUNT               1     //00000001
#define         I86_PIT_OCW_MASK_MODE                   0xE  //00001110
#define         I86_PIT_OCW_MASK_RL                 0x30//00110000
#define         I86_PIT_OCW_MASK_COUNTER                0xC0      //11000000
```

Okay...While this is smaller then the ICWs and OCWs we set up in the PIC, this is actually more complex. The commands used in the PIC are simple in that they are 1 bit in size. The commands used in this operation command word are not.

This is where **Command Control Bits** shine. These help define the different settings and bit combinations for the different bit masks above. Here they are.

```
#define         I86_PIT_OCW_BINCOUNT_BINARY   0     //0         //! Use when setting I86_PIT_OCW_MASK_BINCOUNT
#define         I86_PIT_OCW_BINCOUNT_BCD 1     //1
#define         I86_PIT_OCW_MODE_TERMINALCOUNT   0     //0000          //! Use when setting I86_PIT_OCW_MASK_MODE
#define         I86_PIT_OCW_MODE_ONESHOT      0x2  //0010
#define         I86_PIT_OCW_MODE_RATEGEN      0x4  //0100
#define         I86_PIT_OCW_MODE_SQUAREWAVEGEN 0x6  //0110
#define         I86_PIT_OCW_MODE_SOFTWARETRIG     0x8  //1000
#define         I86_PIT_OCW_MODE_HARDWARETRIG    0xA  //1010
#define         I86_PIT_OCW_RL_LATCH        0     //000000  //! Use when setting I86_PIT_OCW_MASK_RL
#define         I86_PIT_OCW_RL_LSBONLY          0x10//010000
#define         I86_PIT_OCW_RL_MSBONLY          0x20//100000
#define         I86_PIT_OCW_RL_DATA         0x30//110000
#define         I86_PIT_OCW_COUNTER_0           0     //00000000     //! Use when setting I86_PIT_OCW_MASK_COUNTER
#define         I86_PIT_OCW_COUNTER_1           0x40//01000000
#define         I86_PIT_OCW_COUNTER_2           0x80//10000000
```

Lets look at an example. Lets say we want to initialize counter 0 as a square wave generator in binary count mode. This is how we can do it:

```
    uint8_t ocw=0;
    ocw = (ocw & ~I86_PIT_OCW_MASK_MODE) | I86_PIT_OCW_MODE_SQUAREWAVEGEN;
    ocw = (ocw & ~I86_PIT_OCW_MASK_BINCOUNT) | I86_PIT_OCW_BINCOUNT_BINARY;
    ocw = (ocw & ~I86_PIT_OCW_MASK_COUNTER) | I86_PIT_OCW_COUNTER_0;
```

I think I am making this too easy, what do you think? :p This is all you need to do, and **ocw** will contain the operation command word that can be sent to the PIC. Notice how using these constants help both improve readabilty, but also to decrease the possibility for errors.

I guess thats all there is to pit.h. Lets dive into pit.cpp next, shall we? Wee...!!

pit.cpp: Implimentation

This containes the bulk of the PIT minidriver. It containes the implimentations of each routine used by both the interface and implimentation.

pit.cpp: Registers

This is where we define the constants to abstract the port locations for the PIT.

```
#define         I86_PIT_REG_COUNTER0        0x40
#define         I86_PIT_REG_COUNTER1        0x41
#define         I86_PIT_REG_COUNTER2        0x42
#define         I86_PIT_REG_COMMAND         0x43

//! Global Tick count
uint32_t        _pit_ticks=0;
```

Not to bad. **I86_PIT_REG_COUNTER0, I86_PIT_REG_COUNTER1,** and **I86_PIT_REG_COUNTER2** are the data registers for each counter. Remember that the PIT has three internal counters? **I86_PIT_REG_COMMAND** is our command register. We will need to write commands to the command register to control and operate the PIT.

Also, notice **_pit_ticks**. This is a very special and important global.

Remember that the PIT counter 0 connects to the IR0 line on the PIC? This means, when counter 0 fires, it will generate **Interrupt Request (IRQ) 0**. We will need to create and install an interrupt handler to handle this request.

All the interrupt handler needs to do is update the **Global Tick Count** for the system. That is what **_pit_ticks** is for.

i86_pit_irq(): PIT Counter 0 Interrupt Handler

This is the interrupt handler that handles the IRQ 0 request. Whenever Counter 0 fires, it will call this interrupt handler.

All it does is increment the global tick count whenever it fires. Note the general format for an interrupt handler.

intstart() is a macro used to disable hardware interrupts and save the stack frame so that we can return to the task without missing up its stack. **intret()** is a macro that disables hardware interrupts, restors the stack frame and returns from the handler using the **IRETD** instruction. The purpose of this is simply so that we can protect the current stack from being changed, and return back to the task with its stack intact. These macros are defined in **asm/system.h** so they can be used by the kernel and device drivers interrupt handlers.

interrupt is a special constant that is only used on certain compiliers. For MSVC++, it is defined as **__declspec (naked)**. This is so we don't need to worry about the compiliers added code. Some compiliers support this keyword directly (Most notably 16 bit compiliers). Others (Like MSVC++) do not, so we must define it.

interruptdone() is a special routine defined in the **Hardware Abstraction Layer**. It is resposible for sending the **End of Interrupt** commands to the PIC.

This is the generic format that all of our interrupt handlers will use.

```
void interrupt _cdecl i86_pit_irq () {

	//! macro to hide interrupt start code
	intstart ();

	//! increment tick count
	_pit_ticks++;

	//! tell hal we are done
	interruptdone(0);

	//! macro used with intstart to return from interrupt handler
	intret ();
}
```

i86_pit_send_command (): Send Command to PIT

This is a very important routine that allows us to send command to the PIT. This hides the command port we are sending it to, which is nice if we need to change the port name. The command is in the form of an **Operation Command Word (OCW)**.

```
//! send command to pic
void i86_pit_send_command (uint8_t cmd) {

	outportb (I86_PIT_REG_COMMAND, cmd);
}
```

For an example, we can build up an OCW using our bit masks and command control bits above. Then, we can use **i86_pit_send_command()** to send the OCW to the PIT.

i86_pit_send_data() and i86_pit_read_data(): Sends and reads data from counter

These routines help abstract the port name used when reading or writing to a counter. These are used to set and get the current count value. All

they do is test the counter passed in **counter** to insure we get the correct port. Then, its just a simple read or write operation through that port.

```
//! send data to a counter
void i86_pit_send_data (uint16_t data, uint8_t counter) {

	uint8_t	port= (counter==I86_PIT_OCW_COUNTER_0) ? I86_PIT_REG_COUNTER0 :
		((counter==I86_PIT_OCW_COUNTER_1) ? I86_PIT_REG_COUNTER1 : I86_PIT_REG_COUNTER2);

	outportb (port, data);
}

//! read data from counter
uint8_t i86_pit_read_data (uint16_t counter) {

	uint8_t	port= (counter==I86_PIT_OCW_COUNTER_0) ? I86_PIT_REG_COUNTER0 :
		((counter==I86_PIT_OCW_COUNTER_1) ? I86_PIT_REG_COUNTER1 : I86_PIT_REG_COUNTER2);

	return inportb (port);
}
```

i86_pit_initialize (): Initialize the PIT

Okay, lets talk about initializing the PIT. Yeah! Well... er.. There really is not much to talk about, as it really does not require initialization. What we will need to do, however, is provide a way to install our interrupt handler. **irq** is the interrupt number to use and **irCodeSeg** is the **code seletor** offset into the **Global Descriptor Table (GDT)**.

We use our **i86_install_ir()** routine to install our interrupt handler (**i86_pit_irq**) into the **Interrupt Descriptor Table**. From here on out, IRQ 0 is mapped to our interrupt handler at **irq**. **irq** should be the same base IRQ number that the primary PIC was mapped to use to insure it is mapped to IRQ 0.

```
//! initialize minidriver
void i86_pit_initialize (uint8_t irq, uint8_t irCodeSeg) {

	//! Install our interrupt handler
	i86_install_ir (irq, I86_IDT_DESC_PRESENT | I86_IDT_DESC_BIT32,
		irCodeSeg, i86_pit_irq);
}
```

i86_pit_start_counter (): Starts an internal counter

This is the final routine in the PIT interface. This starts up a counter. We pass the counter into **counter** that we want to start (Such as

I86_PIT_REG_COUNTER0). **mode** containes the operation mode that we want the counter to use (Such as **I86_PIT_OCW_MODE_SQUAREWAVEGEN**). **freq** containes the frequency rate that we want the counter to operate at.

This routine builds up the **operational command word** based on the paramaters passed into the routine.

```
void i86_pit_start_counter (uint32_t freq, uint8_t counter, uint8_t mode) {

	if (freq==0)
		return;

	uint16_t divisor = 1193180 / freq;

	//! send operational command
	uint8_t ocw=0;
	ocw = (ocw & ~I86_PIT_OCW_MASK_MODE) | mode;
	ocw = (ocw & ~I86_PIT_OCW_MASK_RL) | I86_PIT_OCW_RL_DATA;
	ocw = (ocw & ~I86_PIT_OCW_MASK_COUNTER) | counter;
	i86_pit_send_command (ocw);

	//! set frequency rate
	i86_pit_send_data (divisor & 0xff, 0);
	i86_pit_send_data ((divisor >> 8) & 0xff, 0);

	//! reset tick count
	_pit_ticks=0;
}
```

Conclusion

From here on out, all of the basics are completed. We have covered alot in this series, from processor modes and architecture, to processor tables, interrupts, interrupt management, and more. This is the beginning of the kernel, and where the kernel builds off from.

In this tutorial, we have added support for the PIC, PIT, exceptions, and hardware interrupt management. This is a important steps, as alot of important devices use hardware interrupts. Also, this gives us a chanch to re-enable hardware interrupts (Remember that we needed to disable hardware interrupts before the switch to protected mode?)

In the next tutorial, we will go back to the kernel itself. Its time to talk about one of the most fundemental aspects of any computer system: **Paging** and **Low Level Memory Management**. This will also be the foundation of our own **System API**.

I'll 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!

 Chapter 15 Home Chapter 17