Interrupts, Exceptions, and IDTs

Part 1 - Interrupts, ISRs, IRQs, and the PIC

What is an Interrupt?

Intel defines an interrupt as, "They[interrupts] alter the normal program flow to handle external events or to report errors or exceptional conditions".

This definition leaves out some very important information. Namely, how does an interrupt happen? An interrupt may be generated either via software or hardware. For instance, when you hit a key on the keyboard, and interrupt is signalled to the CPU. If everything is set up correctly, the CPU will stop the currently running code and call a function that will read port 0x60(the keyboard's output port) to find out what the keyboard is sending. This function should then return control to whatever was running before the keyboard caused the interrupt. Often, whatever orignal code was executing never knows that an interrupt happened. Software can also trigger interrupts via the assembly instruction, 'int <interrupt number>'.

Just a quick sidenote, you can enable and disable interrupts with the 'cli'(disable) and 'sti'(enable) assembly instructions.

How can Interrupts be Used?

As I mentioned already, the keyboard can trigger an interrupt when it's sending information such as a keypress. Many other hardware devices use interrupts also. Hard drives, floppy drives, soundcards, CD-ROM drives, and network cards all use interrupts to either tell the operating system that they have completed something or have data to give to the operating system. The PIT(**P**rogramable **I**nterrupt **T**imer) also triggers an interrupt at a specified interval, this makes the PIT useful for preemptive multitasking(but that's a topic for another tutorial).

User programs can also use interrupts. MS-DOS provides several interrupts to programs so data can be printed to the screen. The BIOS supplies several interrupts(these interrupts only work in Real Mode) for programs to switch between graphics modes.

So What's an ISR?

An ISR(**I**nterrupt **S**ervice **R**outine) is the code executed when an interrupt occurs. There is one ISR per interrupt and it's your job to code the ISRs. How the CPU figures out which ISR to execute is covered in Part 3, IDTs. Now for the exciting part, how to code an ISR!

Coding an ISR

At first, the concept of coding a function that must not tamper with a currently running process seems somewhat difficult. It's not actually. Fortunately the CPU takes care of the most compilcated part by saving the SS, EIP, ESP, and CS registers to the stack. As long as the stack pointer is still the same at the start of the ISR and right before a 'iret' instruction, 'iret' will restore the registers back for you automatically. So, all you are left with is the job of making sure that if you mess with EAX, EBX, ECX, EDX, EBP, ESI, EDI, ES, DS, FS, or GS, that you save them first, and then restore them once you are done using them.

So how does this look? Here's a template in NASM assembly:

```
isr:
 pusha
 push gs
 push fs
 push es
 push ds

 ; do what you want to here :)

 pop ds
 pop es
 pop fs
 pop gs
 popa
 iret
```

IRQs

IRQs(**I**nterrupt **Req**uest) are interrupts triggered by the hardware in a computer.

There are 16 IRQs total on the PC numbered 0-15. The PIC(**P**rogramable **I**nterrupt **C**ontroller) maps these IRQs in one two blocks of 8 IRQs each to two blocks of interrupts. By default, the first 8 IRQs are mapped interrupts 8-15 and the last 8 are mapped to interrupts 112-119. This interferes with exceptions(covered later in detail), so we need to remap the IRQs to a different block of interrupt numbers. I'm not going to tell you how to do this, so I instead suggest you read the info here.

More in-depth info on IRQs may be found in the IRQs tutorial by Ralph Griffin.

The PIC, the Glue Between IRQs and the CPU

When a hardware interrupt occurs, there's a lot of stuff that has to happen. To simplify this for hardware vendors(and provide better compatiblity for devices between plateforms), when a device triggers an IRQ, it tells the PIC(**P**rogramable **I**nterrupt **C**ontroller) about it and gives all the info needed to the PIC. The PIC figures out what interrupt number the IRQ corrosponds to, then the PIC signals the CPU that an IRQ has occured. Once the CPU finishes the current instruction, the CPU runs the interrupt number the PIC gives it.

Thanks go to Jimferd/Akira for proofreading