IRQs and PICs

Interrupt Requests or IRQs are interrupts that are raised by hardware devices. Some devices generate an IRQ when they have data ready to be read, or when they finish a command like writing a buffer to disk, for example. It's safe to say that a device will generate an IRQ whenever it wants the processor's attention. IRQs are generated by everything from network cards and sound cards to your mouse, keyboard, and serial ports.

Any IBM PC/AT Compatible computer (anything with a 286 and later processor) has 2 chips that are used to manage IRQs. These 2 chips are known as the Programmable Interrupt Controllers or PICs. These PICs also go by the name '8259'. One 8259 acts as the 'Master' IRQ controller, and one is the 'Slave' IRQ controller. The slave is connected to IRQ2 on the master controller. The master IRQ controller is connected directly to the processor itself, to send signals. Each PIC can handle 8 IRQs. The master PIC handles IRQs 0 to 7, and the slave PIC handles IRQs 8 to 15. Remember that the slave controller is connected to the primary controller through IRQ2: This means that every time an IRQ from 8 to 15 occurs, IRQ2 fires at exactly the same time.

When a device signals an IRQ, remember that an interrupt is generated, and the CPU pauses whatever it's doing to call the ISR to handle the corresponding IRQ. The CPU then performs whatever necessary action (like reading from the keyboard, for example), and then it must tell the PIC that the interrupt came from that the CPU has finished executing the correct routine. The CPU tells the right PIC that the interrupt is complete by writing the command byte 0x20 in hex to the command register for that PIC. The master PIC's command register exists at I/O port 0x20, while the slave PIC's command register exists at I/O port 0xA0.

Before we get into writing our IRQ management code, we need to also know that IRQ0 to IRQ7 are originally mapped to IDT entries 8 through 15. IRQ8 to IRQ15 are mapped to IDT entries 0x70 through 0x78. If you remember the previous section of this tutorial, IDT entries 0 through 31 are reserved for exceptions. Fortunately, the Interrupt Controllers are 'programmable': You can change what IDT entries that their IRQs are mapped to. For this tutorial, we will map IRQ0 through IRQ15 to IDT entries 32 through 47. To start us off, we must add some ISRs to 'start.asm' in order to service our interrupts:

```
global _irq0
...                ; You complete the rest!
global _irq15

; 32: IRQ0
_irq0:
    cli
    push byte 0    ; Note that these don't push an error code on the stack:
                   ; We need to push a dummy error code
    push byte 32
    jmp irq_common_stub

...                ; You need to fill in the rest!

; 47: IRQ15
_irq15:
    cli
    push byte 0
    push byte 47
    jmp irq_common_stub
```

```
extern _irq_handler

; This is a stub that we have created for IRQ based ISRs. This calls
; '_irq_handler' in our C code. We need to create this in an 'irq.c'
irq_common_stub:
    pusha
    push ds
    push es
    push fs
    push gs
    mov ax, 0x10
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov eax, esp
    push eax
    mov eax, _irq_handler
    call eax
    pop eax
    pop gs
    pop fs
    pop es
    pop ds
    popa
    add esp, 8
    iret
```

Add this chunk of code to 'start.asm'

Just like each section of this tutorial before this one, we need to create a new file called 'irq.c'. Edit 'build.bat' to add the appropriate line to get GCC to compile to source, and also remember to add a new object file to get LD to link into our kernel.

```
#include < system.h >

/* These are own ISRs that point to our special IRQ handler
*  instead of the regular 'fault_handler' function */
extern void irq0();
...                     /* Add the rest of the entries here to complete the declarations */
extern void irq15();

/* This array is actually an array of function pointers. We use
*  this to handle custom IRQ handlers for a given IRQ */
void *irq_routines[16] =
{
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0
};

/* This installs a custom IRQ handler for the given IRQ */
void irq_install_handler(int irq, void (*handler)(struct regs *r))
{
    irq_routines[irq] = handler;
}

/* This clears the handler for a given IRQ */
void irq_uninstall_handler(int irq)
{
    irq_routines[irq] = 0;
}

/* Normally, IRQs 0 to 7 are mapped to entries 8 to 15. This
*  is a problem in protected mode, because IDT entry 8 is a
*  Double Fault! Without remapping, every time IRQ0 fires,
*  you get a Double Fault Exception, which is NOT actually
```

```
*  what's happening. We send commands to the Programmable
*  Interrupt Controller (PICs - also called the 8259's) in
*  order to make IRQ0 to 15 be remapped to IDT entries 32 to
*  47 */
void irq_remap(void)
{
    outportb(0x20, 0x11);
    outportb(0xA0, 0x11);
    outportb(0x21, 0x20);
    outportb(0xA1, 0x28);
    outportb(0x21, 0x04);
    outportb(0xA1, 0x02);
    outportb(0x21, 0x01);
    outportb(0xA1, 0x01);
    outportb(0x21, 0x0);
    outportb(0xA1, 0x0);
}

/* We first remap the interrupt controllers, and then we install
*  the appropriate ISRs to the correct entries in the IDT. This
*  is just like installing the exception handlers */
void irq_install()
{
    irq_remap();

    idt_set_gate(32, (unsigned)irq0, 0x08, 0x8E);
    ...           /* You need to add the rest! */
    idt_set_gate(47, (unsigned)irq15, 0x08, 0x8E);
}

/* Each of the IRQ ISRs point to this function, rather than
*  the 'fault_handler' in 'isrs.c'. The IRQ Controllers need
*  to be told when you are done servicing them, so you need
*  to send them an "End of Interrupt" command (0x20). There
*  are two 8259 chips: The first exists at 0x20, the second
*  exists at 0xA0. If the second controller (an IRQ from 8 to
*  15) gets an interrupt, you need to acknowledge the
*  interrupt at BOTH controllers, otherwise, you only send
*  an EOI command to the first controller. If you don't send
*  an EOI, you won't raise any more IRQs */
void irq_handler(struct regs *r)
{
    /* This is a blank function pointer */
    void (*handler)(struct regs *r);

    /* Find out if we have a custom handler to run for this
    *  IRQ, and then finally, run it */
    handler = irq_routines[r->int_no - 32];
    if (handler)
    {
        handler(r);
    }

    /* If the IDT entry that was invoked was greater than 40
    *  (meaning IRQ8 - 15), then we need to send an EOI to
    *  the slave controller */
    if (r->int_no >= 40)
    {
        outportb(0xA0, 0x20);
    }

    /* In either case, we need to send an EOI to the master
    *  interrupt controller too */
    outportb(0x20, 0x20);
}
```

The contents of 'irq.c'

In order to actually install the IRQ handling ISRs, we need to call 'irq_install' from inside the 'main' function in 'main.c'. Before you add the call, you need to add function prototypes to 'system.h' for 'irq_install', 'irq_install_handler', and 'irq_uninstall_handler'. 'irq_install_handler' is used for allowing us to install a special custom IRQ sub handler for our device under a given IRQ. In a later section, we will use 'irq_install_handler' to install a custom IRQ handler for both the System Clock (The PIT - IRQ0) and the Keyboard (IRQ1). Add 'irq_install' to the 'main' function in 'main.c', right after we install our exception ISRs. Immediately following that line, it's safe to allow IRQs to happen. Add the line: __asm__ __volatile__ ("sti");

Congratulations, you have now followed how to step by step create a simple kernel that is capable of handling IRQs and Exceptions. An IDT is installed, along with a custom GDT to replace the original one loaded by GRUB. If you have understood all that is mentioned up until this point, you have passed one of the biggest hurdles associated with Operating System development. Most hobbyist OS developers do not successfully get past installing ISRs and an IDT. Next, we will learn about the simplest device to use an IRQ: The Programmable Interval Timer (PIT).