The PIT: A System Clock

The Programmable Interval Timer (PIT, model 8253 or 8254), also called the System Clock, is a very useful chip for accurately generating interrupts at regular time intervals. The chip itself has 3 channels: Channel 0 is tied to is tied to IRQ0, to interrupt the CPU at predictable and regular times, Channel 1 is system specific, and Channel 2 is connected to the system speaker. As you can see, this single chip offers several very important services to the system.

The only channels that you should every be concerned with are Channels 0 and 2. You may use Channel 2 in order to make the computer beep. In this section of the tutorial, we are only concerned with Channel 0 - mapped to IRQ0. This single channel of the timer will allow you to accurately schedule new processes later on, as well as allow the current task to wait for a certain period of time (as will be demonstrated shortly). By default, this channel of the timer is set to generate an IRQ0 18.222 times per second. It is the IBM PC/AT BIOS that defaults it to this. A reader of this tutorial has informed me that this 18.222Hz tick rate was used in order for the tick count to cycle at 0.055 seconds. Using a 16-bit timer tick counter, the counter will overflow and wrap around to 0 once every hour.

To set the rate at which channel 0 of the timer fires off an IRQ0, we must use our outportb function to write to I/O ports. There is a Data register for each of the timer's 3 channels at 0x40, 0x41, and 0x42 respectively, and a Command register at 0x43. The data rate is actually a 'divisor' register for this device. The timer will divide it's input clock of 1.19MHz (1193180Hz) by the number you give it in the data register to figure out how many times per second to fire the signal for that channel. You must first select the channel that we want to update using the command register before writing to the data/divisor register. What is shown in the following two tables is the bit definitions for the command register, as well as some timer modes.

7 6	5 4	3 1	0
CNTR	RW	Mode	BCD

CNTR - Counter # (0-2)
RW - Read Write mode
(1 = LSB, 2 = MSB, 3 = LSB then MSB)
Mode - See right table
BCD - (0 = 16-bit counter,
1 = 4x BCD decade counters)

Mode	Description
0	Interrupt on terminal count
1	Hardware Retriggerable one shot
2	Rate Generator
3	Square Wave Mode
4	Software Strobe
5	Hardware Strobe

Bit definitions for 8253 and 8254 chip's Command Register located at 0x43

To set channel 0's Data register, we need to select counter 0 and some modes in the Command register first. The divisor value we want to write to the Data register is a 16-bit value, so we will need to transfer both the MSB (Most Significant Byte) and LSB (Least Significant Byte) to the data register. This is a 16-bit value, we aren't sending data in BCD (Binary Coded Decimal), so the BCD field should be set to 0. Finally, we want to generate a Square Wave: Mode 3. The resultant byte that we should set in the Command register is 0x36. The above 2 paragraphs and tables can be summed up into this function. Use it if you wish, we won't use it in this tutorial to keep things simple. For accurate and easy timekeeping, I recommend setting to 100Hz in a real kernel.

```
void timer_phase(int hz)
{
    int divisor = 1193180 / hz;       /* Calculate our divisor */
    outportb(0x43, 0x36);             /* Set our command byte 0x36 */
    outportb(0x40, divisor & 0xFF);   /* Set low byte of divisor */
    outportb(0x40, divisor >> 8);     /* Set high byte of divisor */
}
```

Not bad, eh?

Create a file called 'timer.c', and add it to your 'build.bat' as you've been shown in the previous sections of this tutorial. As you analyse the following code, you will see that we keep track of the amount of ticks that the timer has fired. This can be used as a 'system uptime counter' as your kernel gets more complicated. The timer interrupt here simply uses the default 18.222Hz to figure out when it should display a simple "One second has passed" message every second. If you decide to use the 'timer_phase' function in your code, you should change the 'timer_ticks % 18 == 0' line in 'timer_handler' to 'timer_ticks % 100 == 0' instead. You could set the timer phase from any function in the kernel, however I recommend setting it in 'timer_install' if anything, to keep things organized.

```
#include < system.h >

/* This will keep track of how many ticks that the system
*  has been running for */
int timer_ticks = 0;

/* Handles the timer. In this case, it's very simple: We
*  increment the 'timer_ticks' variable every time the
*  timer fires. By default, the timer fires 18.222 times
*  per second. Why 18.222Hz? Some engineer at IBM must've
*  been smoking something funky */
void timer_handler(struct regs *r)
{
    /* Increment our 'tick count' */
    timer_ticks++;

    /* Every 18 clocks (approximately 1 second), we will
    *  display a message on the screen */
    if (timer_ticks % 18 == 0)
    {
        puts("One second has passed\n");
    }
}

/* Sets up the system clock by installing the timer handler
*  into IRQ0 */
void timer_install()
{
    /* Installs 'timer_handler' to IRQ0 */
    irq_install_handler(0, timer_handler);
}
```

Example of using the system timer: 'timer.c'

Remember to add a call to 'timer_install' in the 'main' function in 'main.c'. Having trouble? Remember to add a function prototype of 'timer_install' to 'system.h'! The next bit of code is more of a demonstration of what you can do with the system timer. If you look carefully, this simple function waits in a loop until the given time in 'ticks' or timer phases has gone by. This is almost the same as the standard C library's function 'delay', depending on your timer phase that you set:

```
/* This will continuously loop until the given time has
*  been reached */
void timer_wait(int ticks)
{
    unsigned long eticks;

    eticks = timer_ticks + ticks;
    while(timer_ticks < eticks);
}
```

If you wish, add this to 'timer.c' and a prototype to 'system.h'

Next, we will discuss how to use the keyboard. This involves installing a custom IRQ handler just like this tutorial, with hardware I/O on each interrupt.