RTC

From OSDev Wiki

Contents

Introduction

A typical OS will use the APIC or PIT for timing purposes. However, the RTC works just as well. RTC stands for Real Time Clock. It is the chip that keeps your computer's clock up-to-date. Within the chip is also the 64 bytes of CMOS RAM.

If you simply want information about reading the date/time from the RTC, then please see the CMOS article. The rest of this article covers the use of RTC interrupts.

Capabilities

The RTC is capable of multiple frequencies. The base frequency is pre-programmed at 32.768 kHz. It is possible to change this value, but this is the only base frequency that will keep proper time. For this reason, it is strongly recommended that you NOT change the base frequency. The chip also has a "divider" register that will generate other frequencies from the base frequency. The output (interrupt) divider frequency is by default set so that there is an interrupt rate of 1024 Hz. If you need an interrupt frequency other than 1024 Hz, the RTC can theoretically generate 15 interrupt rates between 2 Hz and 32768 Hz. On most machines however, the RTC interrupt rate can not go higher than 8 kHz.

Programming the RTC

RTC interrupts are disabled by default. If you turn on the RTC interrupts, the RTC will periodically generate IRQ 8.

Avoiding NMI and Other Interrupts While Programming

When programming the RTC, it is important that the NMI (non-maskable-interrupt) and other interrupts are disabled. This is because if an interrupt happens, the RTC may be left in an "undefined" (non functional) state. This would usually not be too big a deal, except for two things. The RTC is never initialized by BIOS, and it is backed up with a battery. So even a cold reboot may not be enough to get the RTC out of an undefined state! See the NMI page for more information about disabling and enabling it, and the effects of it.

Setting the Registers

The 2 IO ports used for the RTC and CMOS are 0x70 and 0x71. Port 0x70 is used to specify an index or "register number", and to disable NMI. Port 0x71 is used to read or write from/to that byte of CMOS configuration space. Only three bytes of CMOS RAM are used to control the RTC periodic interrupt function. They are called RTC Status Register A, B, and C. They are at offset 0xA, 0xB, and 0xC in the CMOS RAM. To write 0x20 to Status Register A you would do this:

```
disable_ints();         // important that no interrupts happen (perform a CLI)
outportb(0x70, 0x8A);   // select Status Register A, and disable NMI (by setting the 0x80 bit)
outportb(0x71, 0x20);   // write to CMOS/RTC RAM
enable_ints();          // (perform an STI) and reenable NMI if you wish
```

Other bytes of the CMOS RAM are used by the RTC for other functions, or by the BIOS and other such services.

IRQ Danger

Since the IRQ number is 8, it has a lower priority in the PIC than the IRQs with lower numbers. While those other interrupts are being handled (until your OS sends an EOI and STI), your OS will not receive any clock ticks. Any IRQ handlers that depend on clock ticks may fail for that reason, because an IRQ of higher number will not preempt an IRQ of lower number.

Turning on IRQ 8

To turn on the periodic interrupt, all you have to do is:

```
disable_ints();                 // disable interrupts
outportb(0x70, 0x8B);           // select register B, and disable NMI
char prev=inportb(0x71);        // read the current value of register B
outportb(0x70, 0x8B);           // set the index again (a read will reset the index to register D)
outportb(0x71, prev | 0x40);    // write the previous value ORed with 0x40. This turns on bit 6 of register B
enable_ints();
```

This will turn on the IRQ with the default 1024 Hz rate. Be sure that you install the IRQ handler before you enable the RTC IRQ. The interrupt will happen almost immediately.

Changing Interrupt Rate

Changing the output divider changes the interrupt rate, *without* interfering with the RTC's ability to keep proper time. The lower 4 bits of register A is the divider value. The default is 0110b, or 6. The setting must be a value from 1 to 15. A value of 0 disables the interrupt. To calculate a new frequency:

```
frequency =  32768 >> (rate-1);
```

"Rate" is the divider setting. If you select a rate of 1 or 2, the RTC will have problems and "roll over" so that it generates interrupts of .81 mS and 3.91 mS, rather than the expected interrupts of 61.0 uS or 30.5 uS. So, the fastest rate you can select is 3. This will generate interrupts of 8 kHz or 122 uS. To change the rate:

```
rate &= 0x0F;                   // rate must be above 2 and not over 15
disable_ints();
outportb(0x70, 0x8A);           // set index to register A, disable NMI
char prev=inportb(0x71);        // get initial value of register A
outportb(0x70, 0x8A);           // reset index to A
outportb(0x71, (prev & 0xF0) | rate); //write only our rate to A. Note, rate is the bottom 4 bits.
enable_ints();
```

Interrupts and Register C

It is important to know that upon a IRQ 8, Status Register C will contain a bitmask telling which interrupt happened. The RTC is capable of producing a periodic interrupt (what this article describes), an update ended interrupt, and an alarm interrupt. If you are only using the RTC as a simple timer this is not important. **What is important is that if register C is not read after an IRQ 8, then the interrupt will not happen again.** So, even if you don't care about what type of interrupt it is, just attach this code to the bottom of your IRQ handler to be sure you get another interrupt.

```
outportb(0x70, 0x0C);   // select register C
inportb(0x71);          // just throw away contents
```

Possible Uses

You could use just the RTC and not use the PIT (the RTC is easier to program in my opinion). My favorite use is to use the RTC for my main kernel clock (controls scheduling and all), and then use the PIT to provide a more accurate wait() function, that can operate on timescales of micro-seconds.

Problems?

Some RTC timer code may not work on some real machines. I have only found about one out of 5 computers that does not work right. The observed problem is a timer tick happened about once every second. I'm not sure why this is, and am trying to find a solution. The machine I observed this on was a 2005 or so Dell (which also has issues USB booting...)

See Also

- CMOS

External Links

- Periodic Interrupts with the Real Time Clock (http://www.compuphase.com/int70.txt)
- CMOS Ram Data Register (http://www.ousob.com/ng/interrupts_and_ports/ng9116b.php)
- Using the 1024 HZ Timer Interrupt (http://www.nondot.org/sabre/os/files/MiscHW/CMOSTimer.html)
- Real Time Clock / CMOS Setup Reference (http://www.nondot.org/sabre/os/files/MiscHW/RealtimeClockFAQ.txt)
- Accessing the CMOS Chip (http://bos.asmhackers.net/docs/timer/docs/cmos.pdf)
- Software interrupt based real time clock source code project for PIC microcontroller (http://users.tkk.fi/~jalapaav/Electronics/Pic/Clock/index.html)

Categories: Common Devices | Interrupts | Time
