Periodic Interrupts with the Real Time Clock

by Thiadmer Riemersma

Introduction

Two previous versions of this technical note have been distributed before. The first one was a small program without any documentation (other than comments in the source), the second version had a decent explanation (I think) about the Real Time Clock hardware, but fell short in describing how the pieces fit together.

This version has expanded coverage on side issues, such as the Peripheral Interface Controllers and on compatibility with the BIOS. In addition, it contains an example in assembly.

Acknowledgements

- Todd Allen (todd.allen@aquila.com) from Avid Software gave feedback on his use of the Real Time Clock.
- Dave Bolt (CIS 100112,552) pointed out that the documentation of the BIOS functions that make use of the Real Time Clock is misleading.
- Steven J. Eschweiler (CIS 73552,2634) urged me to not only talk about the bare metal, but to set a "standard" in the hope that several programs will be able to use the RTC without conflicting with each other.
- Terry Greenaway from AVIX inc. pointed out differences between AMI and Phoenix BIOSes and limitations of the frequency range of the periodic interrupts.
- Peter Hoffman, ???

- Nu-Mega Technologies Inc. for their wonderful system debugger Soft-ICE.

History

The original IBM PC only kept the date and time while it was running. Each time you booted the computer, it reset the date to January 1, 1980, 0:00 hour. At these days, the AUTOEXEC.BAT file usually contained the DATE and TIME commands, so that you were prompted for the proper date at each boot.

The clock that keeps the current time and date is an 8253 or 8254 chip (or compatible) with three programmable counters. Two of these counters are used for DRAM refresh and to drive the speaker. The third one is hooked to IRQ 0 (interrupt 8) and the default handler for interrupt 8 in the system BIOS adjusts the current time in the BIOS data area at each "tick".

Over time, many programs that needed periodic service also hooked interrupt 8. The PRINT.COM program that comes with DOS is a good example. Some programs also require a faster period than the 18.2 ticks/second standard rate (for example, execution profilers). So they reprogram the timer.

Here the problems begin. First of all, the default BIOS ISR must still see 18.2 ticks per second to keep the time straight. For example, when you reprogram the timer to tick at 91 Hz and pass only one out of 5 interrupts to the previous interrupt handler, this one sees 91/5 = 18.2 ticks per second. If the new timer frequency is not a multiple of 18.2, the process is slightly more complicated, but it is still possible to pass 18.2 interrupts per second to the previous handler on the average.

But this trick only works if the ISR that uses the higher interrupt frequency is the last in the chain of interrupt 8 hooks. If another ISR hooks behind it, that one too will see the higher interrupt rate (and possibly draw incorrect conclusions about when to pop up). Worse yet, if that other routine also reprogrammed the timer... In conclusion, when you reprogram the timer in a TSR (where you cannot guarantee

that you are the last in the chain), you may encounter conflicts with other TSRs and application programs. Microsoft Windows has a virtual timer device (VTD) to solve such conflicts between windows applications and DOS boxes.

These problems are what led me to investigate the use of another timer. One that also generates interrupts. One that almost no one uses.

The Real Time Clock

The use of the Real Time Clock is not a general solution to the conflicts outlined above. If enough programs use and/or reprogram the Real Time Clock, we have exactly the same situation. But for now, the use of the Real Time Clock has worked for me in situations that the interrupt 8 method did not.

Why isn't the Real Time Clock more widely used? There may be several reasons. I can think of these:

- The Real Time Clock is not available on all machines. The Real Time Clock was introduced with the IBM PC AT, and most PC XT style computers do not have the circuitry. This means that you also have to detect whether Real Time Clock services are available if you intend to use them.
- The Real Time Clock became available when the interrupt 8 hooking and timer reprogramming had already entered the folklore. People just kept going on the same track.
- The Real Time Clock is less flexible; it handles only 15 possible interrupt rates between 2 Hz and 32767 Hz (theoretical, that is; in practice, the interrupt rate cannot go higher than 8 kHz on most systems).
- The default interrupt handler has the irritating behavior of switching the Real Time Clock interrupt off after a time-out expires.

- The Real Time Clock, and especially the use of the interrupt on IRQ 8 (interrupt 70h), is ill documented. In fact, I found out most of this by diving into the BIOS with Nu-Mega's Soft-ICE debugger.

The RTC/CMOS chip

The chip that contains the Real Time Clock is a Motorola MC146818A CMOS chip. In addition to the Real Time clock, it contains 64 bytes of non-volatile RAM for machine status information. (This is the CMOS data that holds vital information on the type of the hard disk, the amount of memory, etc.)

The RTC/CMOS chip is programmed through the I/O addresses 70h and 71h. Port 70h is the index register and port 71h is the data register. Both ports can be read or written. All internal registers of the RTC/CMOS chip are accessed by setting an index at port 70h and reading from or writing to port 71h. To use the RTC/CMOS chip for periodic interrupts, we only need to consider four of these internal registers: the status registers A to D.

There are a few caveats when programming the RTC/CMOS chip. First of all, the index register is usually set to status register D, and it should remain set to this register. Secondly, after writing to port 70h, you must read from, or write to port 71h. There is no real purpose in writing to port 71h when status register D is selected, because it only indicates whether the RTC/CMOS chip still has power (battery). So after you read or write a value to an internal register, you reset the index register (port 70h) to 0Dh (status register D), and then do a read from port 71h.

Finally, there should not be a long delay between writing to port 70h and reading from or writing to port 71h. Waiting too long between the two operations can cause malfunction of the RTC/CMOS chip. Interrupts must be disabled while programming the RTC. The non-maskable interrupt (NMI) must also be disabled.

The NMI mostly requires you to reboot, so you may be inclined to say "What the heck, why bother to keep the chip in a good shape at an NMI. When an NMI occurs, I have a serious system failure and I'll probably need to do a cold start. So the chip restarts with a clean state anyway". Not so with the RTC/CMOS chip. Remember, this chip continues to work when the computer is switched off. The BIOS does not initialize the chip at boot time; on the contrary, the system reads values from the chip to determine vital parameters such as memory size and the fixed disk type. Malfunction in the RTC/CMOS chip is to be avoided at all cost. That is why you want to avoid an NMI when modifying the RTC, and turn it on afterwards.

Another reason to toggle the NMI is that it is equally simple to turn it off and on as to leave it unchanged. The NMI mask bit is accessed at the same I/O address as the RTC/CMOS index register: port 70h. Bit 7 of port 70h masks or unmasks the NMI at the same time as selecting the index for port 71h.

Now that the basics of the RTC/CMOS programming are covered, we can return to the main course: programming the RTC to generate interrupts.

Status register A is used to select an interrupt rate. The basic oscillator frequency (in the bits 4-6 of status register A) is set to 32,768 Hz. This setting indicates what kind of crystal is connected to the oscillator circuit of the MC146818A chip. On the IBM PC-AT, this is a 32 kHz crystal. Choosing a different frequency setting in status register A can cause the RTC/CMOS chip to update the time incorrectly.

The lower four bits (0-3) of status register A select a divider for the 32,768 Hz frequency. The resulting frequency is used for the periodic interrupt (IRQ 8). The system initializes these bits to 0110 binary (or 6 decimal). This selects a 1,024 Hz frequency (and interrupt rate). Setting the lowest four bits to a different value changes the interrupt rate **without** interfering with the time keeping of the RTC/CMOS chip.

The formula to calculate the interrupt rate is:

```
freq = 32768 >> (rate-1)
```

where '>>' stands for the 'shift left' operation and 'rate' is the value of the lower four bits of status register A. This value must be between 1 and 15 (selecting 0 stops the periodic interrupt). The interrupt frequency you can choose is thus always a power of 2 between 2 Hz and 32768 Hz. There is a caveat, however. With an input signal of 32768 Hz, the timer output 'rolls over' if you set the value 'rate' to 2 or 1. Instead of producing periodic interrupts of 61.0 µs or 30.5 µs respectively, they produce 7.81 ms and 3.91 ms interrupts. The fastest interrupt rate you can generate is 8 kHz (122 µs interrupts), by setting 'rate' to 3. Higher frequencies require a higher base frequency, and this would require a different crystal than is installed in the IBM PC-AT.

Status register B contains a number of flags. We are only interested in one of these flags: the "Periodic Interrupt Enable" or PIE bit. Setting this bit is the only thing left to start generating interrupts to IRQ 8. By the way, the choice of IRQ, interrupt and I/O port numbers is very confusing. When IRQ 8 fires, it generates INT 70h. This is just the way the secondary PIC is programmed. It has nothing to do with I/O port 70h, through which the RTC/CMOS chip is programmed. Also, INT 8 belongs to the standard timer, IRQ 8 to the RTC. We have just to be extra precise when talking about this subject.

When an IRQ 8 fires and interrupt 70h is called, status register C holds a bit mask that tells what kind of interrupt occurred: periodic interrupt, alarm interrupt or update ended interrupt. And unless you read status register C, IRQ 8 will not be generated again. This means that you must read status register C inside your ISR for interrupt 70h even when you normally don't care about its contents. Otherwise you will only see a single interrupt.

The bits in status register C are not used in the example program. They are useful when several interrupts of the RTC are connected to IRQ 8. The bits in status register C let you detect what interrupt caused IRQ 8.

The last thing we have to cover before leaving the RTC/CMOS chip is how to detect the chip. Two common methods are to check that the I/O ports are functioning (write a value into one of them and read it back), and to use a BIOS function that also uses the hardware and check the results. Although it looks cleaner, the BIOS function should be used with caution, because it might use different hardware to achieve the desired result. Indeed, the IBM PS/2 model 30, running on an 8086 processor, has Real Time Clock services, but not with the same hardware. The model 30 uses a VLSI gate array for the keyboard controller, mouse controller, real time clock and interrupt controller. There is only one interrupt controller (compatible with the 8259A PIC chip in the PC XT and AT computers), but all functionality is jammed inside the 8 lines of the PIC by connecting the keyboard, the mouse and the Real Time Clock to IRQ 1.

The Peripheral Interrupt Controllers

The Peripheral Interrupt Controller (PIC) converts IRQ signals to hardware interrupts. The original PC had only one PIC (chip 8259A) for eight IRQ lines. These IRQs 0 to 7 are normally redirected to (hardware) interrupts 8 to 0Fh. Many DOS extenders reprogram the primary PIC to relocate the interrupt range, because interrupts 8 to 0Fh collide with the processor exception interrupts.

The IBM PC-AT and compatibles have two PIC chips in a chain. The second PIC is chained to IRQ 2 of the first PIC. The RTC is connected to the first line of the secondary PIC. The IRQ lines of the secondary PIC are usually translated to interrupts 70h to 77h.

We deal with the PICs at two places. The first one is that an ISR for a hardware interrupt should send an "end of interrupt" (EOI) signal to the PICs before returning. Otherwise the PIC will not generate an interrupt for the IRQ or any IRQ of a lower level. In the case of the RTC, we must send a non- specific EOI to both PICs (since they are chained).

Each PIC has an 8 bit mask that disables selected IRQs. Masked IRQs are not passed through to the microprocessor. The BIOS by American Megatrends Inc. (AMI) disables IRQ 8 (bit 0 in the mask of the secondary PIC) at start-up. We need to unmask IRQ 8 when initializing the RTC. Many BIOSes (e.g. those by Phoenix and those found in the IBM PS/1 and PS/2 series) enable IRQ 8 by default, but it doesn't harm to enable it again. Although the secondary PIC is chained to the primary PIC on the IRQ 2 line, it is usually not necessary to reprogram the mask of the first PIC. IRQ 2 is always enabled, since the hard disk and other essential hardware is also connected to the secondary PIC.

Compatibility with the BIOS

The goal is to be able to jump to the previous INT 70h handler after your handler has done its work. This is common practice when writing ISRs. But in the case of the RTC, you have to deal with the irritating feature of the default INT 70h handler, which shuts the RTC down after the time period in the BIOS data area has expired.

My demo program avoids all this by **not** jumping to the previous INT 70h handler. After all, the intent is to show the workings of the RTC, not that of the BIOS.

A simple way to solve this is to disallow the BIOS functions that use the RTC. To do this, you set the byte at address 0040h:00A0h to 1. This value tells the BIOS that the RTC is in use, and makes functions 83h and 86h of interrupt 15h fail (return with carry flag set). Then, before jumping to the previous INT 70h handler, you store at address 0040h:009Bh a double word value that is at least 976. The default interrupt

handler subtracts 976 from the value at that address and halts the RTC periodic interrupts if at drops below zero (976 is the time in microseconds that passes between two invocations of INT 70h if the RTC is ticking at its default frequency of 1024 Hz).

If you require that the BIOS functions 83h and 86h of interrupt 15h keep working, the best solution probably is to take over interrupt 15h. You then need to simulate these functions. That means that you must update the BIOS data area, detect a time-out and handle accordingly. If you choose this path, remind that the default INT 70h handler also issues INT 4Ah on time-out (in addition to shutting the RTC periodic interrupt down).

Strapping it all together (the program)

I fear of viruses, I only distribute source. You will need an assembler to get the demonstration program running. Considering that you are probably a low level programmer if you are interested in this, I hope this is not too inconvenient.

The demonstration program has been tested on the following hardware:

- 80486 equipped IBM compatible ("Exec") with AMI BIOS.
- IBM PS/2 model 50Z
- IBM PS/1 (model ???)
- Everex Step 486 (with a modified version of AMI BIOS)
- some clone with a Phoenix BIOS (as reported by Terry Greenaway)

It has also been tested with the following software environments:

- works correctly with DOS 5.0

- does **not** work in a DOS box of OS/2 2.1

- works in a DOS box of Windows 3.1, but in a windowed DOS box you will not see interrupts when moving the mouse. Even in a full screen DOS box, you are limited to frequencies below 1 kHz (when running on a 80486/33 MHz).

Now that we have finished our journey through the peculiarities of the Real Time Clock, it is time to discuss where it can be used.

What led me to investigate the RTC, were audio drivers for an audio product that my company makes. This device connects to the printer port; the printer itself is chained on the back of the plug. The concept is very similar to that of the Disney Sound Source.

To generate sound on the device, I had to push a sample through the printer port 8000 to 22000 times per second. The first drivers did this in the foreground by polling (even without reprogramming, you can get very accurate timing read-outs from the 8253 chip, but that's a different story). Later drivers played sound in the background and were, therefore, interrupt driven. I first used interrupt 8h and a reprogrammed 8253. Although there were some compatibility problems, on the whole it worked reasonably well.

The real problem was the Microsoft Windows drivers. Windows virtualizes the 8253 timer chip, meaning that the chip cannot be touched from the outside. It is protected. There is one loophole, called a VxD (short for Virtual Device Driver). If you write a VxD, you can modify the timer, but Microsoft strongly warns against such practices and I feared Windows' instability might be blamed on my driver if it did these tricks. (This is not far fetched. I already got several reports from a customer that their machine hangs at times when my driver, based on the RTC, is installed. The machine also hangs periodically when the driver is not installed, but then, apparently, less frequent. With quite a lot of talking over the telephone, I traced down two of the three bug reports: one was due to a third party DLL that they

use, the second to a faulty screen driver. In all fairness I must say that I also corrected a bug in my driver, but one that was never reported.) So I was looking for other timers, and I was convinced that I would need to a one in the audio plug. The stories I heard about IRQ conflicts with parallel ports and cheap multi-I/O boards that do not handle the IRQ line correctly were not very encouraging. We would only be able to sell these plugs if the installation was effortless. Always!

So you can imagine my excitement when you came upon a remark in one of the IBM Technical References that said the "BIOS sets the Periodic Interrupt rate of the Real Time Clock to 1024 ticks per second". Wait a minute, would that mean that you can achieve higher interrupt frequencies?

You know the rest.

Other people have also used the Periodic Interrupt for the Real Time Clock, and sent me a message (and I highly appreciate that, thank you). Here is what the Real Time Clock is used for:

- Todd Allen from Avid Software developed a paint/animation package called CutOut. CutOut is a 32-bit multi-threaded DOS application which supports painting and multiple concurrent animations. It operates in 8, 16 and 24 bit/pixel modes and can change resolutions on the fly. The screen is a window on a much larger world. The RTC provides a time base for animation, music, and synching to the V-blanking of the monitor. (todd.allen@aquila.com)

- David Bolt uses the BIOS functions for timer services (that ultimately use the Real Time Clock) to drive a stepper motor.

- Steve Eschweiler writes a flight simulator and uses the RTC to avoid conflicts with his audio playback code and other programs/subroutines that change the clock rate of 8253 timer chip.

Author: Thiadmer Riemersma

Email: thiadmer@compuphase.com

Website: http://www.compuphase.com/

Released: Mar 1994