Timer Interrupt Programming

Intro

The timer interrupt is used when a certain event MUST happen at a given frequency. In the Programming the Sound Blaster 16 Example #4 we used it to play back a sample in Direct Mode so that the sample frequency matched the timer interrupt's frequency. This capability not only extremely useful, but very flexible. We can call any function at whatever frequency! Many games use the timer interrupt to move sprites, check collisions and other timely events.

Port Listing

Here's a listing of the ports that we can communicate with to set the correct timing frequency and how we intend on communicating with the Timer Interrupt.

IO Port	Counter #	Usage
40h	Counter 0	Timer / Disk
41h	Counter 1	Memory Refresh
42h	Counter 2	Tape Drive / Speaker
43h	Control Register	Controlling Timer functions

8253 Control Bit Descriptions

The Timer Interrupt is officially called the 8253 in case you were wondering. It can be controlled through its control port which is 0x43. The byte of information that we send there tells it exactly how to behave and how we are going to communicate with it. All we need to know is that writing a byte to port 43h, the Control Register, will set methods on how we want the Timer Interrupt to behave. Any writes to the Counter Ports will set their frequencies! Lets create a function that will replace the current timer interrupt for Counter 0, with a new one.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Function
Counting Method								
							0	Count in binary
							1	Count in BCD
Counter Mode								
				0	0	0		Interrupt on terminal count
				0	0	1		Hardware retriggerable one-shot
				-	1	0		Rate Generator
				-	1	1		Square Wave
				1	0	0		Software retriggerable strobe
				1	0	1		Hardware retriggerable strobe
Read and Write Control								
		1	0					Counter latch operation
		0	1					Read / Write least Significant byte of counter
		1	0					Read / Write most Significant byte of counter
		1	1					Read / Write least, then most, Significant byte
Counter to be accessed for operation								
0	0							0
0	1							1
1	0							2
1	1							Illegal

Hooking the Interrupt

Lets design a function that will replace the current timer interrupt function with a new one to do our bidding! Here's just such a function!

Located in Header file

```
#define TIMERISR        0x1C
#define LOW_BYTE(n) (n & 0x00ff)
#define HIGH_BYTE(n) ((n>>8)&0x00ff)
extern Timer_ISR(_go32_dpmi_registers*);
_go32_dpmi_seginfo OldTimer,NewTimer;
_go32_dpmi_registers timerregs;
```

Located in the CPP file

```
void ISRS::SetupTimerISR()
{ SetTimer=1;
  _go32_dpmi_get_protected_mode_interrupt_vector(TIMERISR,&OldTimer);
  NewTimer.pm_offset = (int)Timer_ISR;
  NewTimer.pm_selector=_go32_my_cs();
  _go32_dpmi_allocate_iret_wrapper(&NewTimer);
  _go32_dpmi_set_protected_mode_interrupt_vector(TIMERISR,&NewTimer);
}
```

Normal C++

Located in Header File

```
#define TIMERISR       0x1C
#define LOW_BYTE(n) (n & 0x00ff)
#define HIGH_BYTE(n) ((n>>8)&0x00ff)
extern void interrupt Timer_ISR(__CPPARGS);
void interrupt (*Old_TIMERISR)(__CPPARGs);
```

Located in the CPP File

```
void ISRS::SetupTimerISR()
{ SetTimer=1;
  Old_TIMERISR = _dos_getvect(TIMERISR);
  _dos_setvect(TIMERISR,Timer_ISR);
}
```

Here we define TIMERISR to be 0x1c which is the correct interrupt number. We also define two macros that will return the High and Low portions of a 16 bit number. We declare Timer_ISR(..) to be extern so that we can define it later on. Remember that we must have it defined before trying to create an instance of our ISRS class. Normal C users have little to worry about, except for not being cool :) In the main SetupTimerISR function we set a variable telling the program that the Timer Interrupt has been altered so that it will be changed back to normal after the program is finished running. The rest you will see a lot when hooking interrupts with DJGPP. We get the pointer to the old ISR, set the offset and selector accordingly, allocate a wrapper to help us out, and whaaamo set the ISR to our new one. Remember that our ISR routine has to be declared globally for this to work at all. Look at the ISR stuff in the sb16e4.zip in the file section to see what the heck I mean! Immediately after this function has run, the new ISR (Timer_ISR) will be called at whatever frequency that timer was set at. By default it is set to 18.2 hz. Now, lets see here, 18.2 doesn't sound real fast does it?! How can we create a fluid moving game with super collision detection and object motion if we can only call those functions 18 times a second?! Luckily we CAN change the frequency of the Timer Interrupt, here's how.

Changing the Frequency

The timer chip is fed with a 1.19318 MHZ clock speed. To set our timer to a given hz, we must take this number and divide it by the hz. As you might see, 1,193,180 / 65535 gives us 18.2 hz!

```
void ISRS::SetTimerFrequency(float hz)
{ float base=1193180.0;
  unsigned short word;
   base/=hz;
   byte=(unsigned short)base;
   outp(0x43,0x3c);
    outp(0x40, LOW_BYTE(word));
    outp(0x40,HIGH_BYTE(word));
}
```

Here's where everything comes together. Our Control Byte must be set as to Count in binary, use Mode 2 (Rate Generator), Read/Write Lease then Most Significant bit, then use Counter 0. So according to our table above it will be 00 111 10 0 with 0x3C being its Hex equivalent. We then write that Control Byte to the Control Register at 0x43. Afterwards we write the Low, then High ends of our frequency to the port for Counter 0 (0x40)! There we have it!

Before Program Exits

Remember that we changed the Interrupt function to ours when we started our program, not to mention its frequency. We must remember to change it back to the original one before our program exits or who knows what will happen!! Since we are going to be using more than just the Timer Interrupt in the ISRS class, lets test the variable before returning it.

```
void ISRS::ReturnISRS()
{
  if(SetTimer)
   {_go32_dpmi_set_protected_mode_interrupt_vector(TIMERISR,&OldTimer);
    _go32_dpmi_free_iret_wrapper(&NewTimer);
    outp(0x43,0x3c);
   outp(0x40,LOW_BYTE(0xFFFF));
   outp(0x40,HIGH_BYTE(0xFFFF));
   }
}
```

And of course Normal C++

```
void ISRS::ReturnISRS()
{
  if(SetTimer)
   {_dos_setvect(TIMERISR,Old_TIMERISR);
    outp(0x43,0x3c);
   outp(0x40,LOW_BYTE(0xFFFF));
   outp(0x40,HIGH_BYTE(0xFFFF));
   }
}
```

After running these functions the Timer ISR will be returned to normal and will be running at the boring 18.2 hz.

NOTE: There are several problems associated with changing the frequency of the Timer Interrupt. First, under Windows95, a program attempting to change the frequency under a DOS Window will be kicked back into Windows and told that the program should run under MS-DOS Mode. Under a Windows DOS Window, I am yet to find any means of changing the frequency. The alternative is to re-boot in MS-DOS Mode and run the program where everything will work fine.

Secondly, when we divide 1.19318 MHZ by our frequency we will get a fractional part left out. For the most part this isn't a problem since most programmers hook up into the Timer Interrupt for the purpose of running collision detection functions or AI algorithms. When the frequency needs to be more precise, things start getting into trouble. In playing sound from a .WAV or .VOC in Direct Mode, the timer must be set precisely at the same frequency as the sample. The result is less than optimum sound quality. More information resides at the Sound Playback in Direct Mode Tutorial should you be utilizing the Timer Interrupt for this purpose.

Thanks for the ear. I think you'll find the Timer Interrupt VERY easy to use and extremely flexible. If you have any questions, comments, rude remarks please give me some feedback!

Contact Information

I just wanted to mention that everything here is copyrighted, feel free to distribute this document to anyone you want, just don't modify it! You can get a hold of me through my website or direct email. Please feel free to email me about anything. I can't guarantee that I'll be of ANY help, but I'll sure give it a try :-)

Email : deltener@mindtremors.com
Webpage : http://www.inversereality.org

Created by
Justin Deltener