5 After This Brief Interruption

Stop me if this hasn't happened to you.

When the clothes dryer buzzed, I decided to take a break. The laundry room light went nova, so I detoured to the garage for a new bulb. On my way through the machine shop, I perched the laundry basket on the workbench while pocketing some outdoor pole lamp parts I'd fixed the previous evening.

I punched the garage door opener, dug a new bulb out of the lamp stash, walked down the driveway to reassemble the widget, then retrieved the day's mailbox treasures. Mary leaned out of the kitchen door to say the call was for me. I passed the mail to her and snagged the phone on the way by.

Several hours later Mary stuck her head in the office and asked, "Why is the garage door open and what have you done with the laundry?"

We pros call this a *blown stack*, although, to be fair, there are other interpretations…

The Firmware Development Board now has enough hardware that we can investigate something that often goes unmentioned: what *really* happens when a hardware interrupt occurs? I'll concentrate on real mode and leave protected mode for a different book.

Our first task must be nailing down the nomenclature. You have probably read about interrupt handlers, exceptions, traps, faults, **IRQ**s, `INT`s, and vectors, but the definitions are often either vague or just plain wrong. After settling those issues, we can examine the code that responds to interrupts. Finally, I'll tell a war story about PC interrupts that should curl your keyboard.

Let's start this discussion at the beginning, all the way down at the bare silicon.

The Inside Story

Intel 80*x*86 CPUs handle interrupts from several sources: external events, instructions, and internal problems such as the dreaded divide-by-zero error. Fortunately, the CPU uses the same basic mechanism in all situations.

External interrupts occur when hardware outside the CPU raises its **INTR** (Interrupt Request) pin, which may happen at any time. The CPU hardware activates its **INTA** (Interrupt Acknowledge) pin and reads a single byte from the data bus. That byte identifies the interrupt source and can, in principle, funnel up

to 256 different external interrupts through the single **INTR** pin. In the PC, as we will see later, only 15 external interrupts contend for the CPU's attention. The CPU ignores **INTR** when the Interrupt Enable bit (a.k.a. the Interrupt Flag or IF) in the FLAGS register is zero.

Unlike external interrupts on the **INTR** pin, Non-Maskable Interrupts arriving at the **NMI** pin cannot be ignored. There is only one Non-Maskable Interrupt, because the CPU does not read an ID byte from the data bus. Hardwired circuitry (or, more precisely, microcode) inside the CPU causes **NMI** to produce INT 02h. Although the CPU cannot disable **NMI** inside the chip (hence the name), the PC system board includes circuitry external to the CPU for that purpose.

Software interrupts occur when the CPU executes one of a class of instructions devoted specifically to causing them. The instructions have mnemonics like INT 10h, INTO, BOUND, and so forth. Like interrupts on the **NMI** pin, software interrupts cannot be ignored. Unlike external interrupts on the **INTR** pin, they are entirely predictable: whenever the CPU executes the instruction, the corresponding software interrupt ensues.

Finally, errors or similar conditions within the CPU trigger exception interrupts. These interrupts are generally data dependent, so a given instruction may not cause an exception every time. However, they can be reproduced if you set all the hardware to precisely the same state. Exceptions cannot be ignored or suppressed.

Each interrupt, regardless of cause, corresponds to a number from 00h through 0FFh (in C, that's 0x00 through 0xFF) called, oddly enough, its Interrupt ID, interrupt type, or just plain interrupt number. By convention, Interrupt IDs use hex notation, although some sources prefer the decimal equivalent.

However, I've seen one reference that managed to convert a *hex* Interrupt ID into *decimal*, then listed the *decimal* value as *hex*. Moral of the story: you must have more than one book on your shelf to crosscheck things that seem out of whack. To help prevent a similar misinterpretation in this book, I'll always indicate that the interrupt ID is in hex: INT 02h or INT $02, as appropriate.

An interrupt's *raison d'être* (you should pardon my French) is diverting the CPU's attention from its current task and setting it to work on something else, something that is, presumably, more important. That something, the code associated with each interrupt, bears the name *interrupt handler* or *interrupt service routine* (a.k.a. ISR). As a general rule, interrupt handlers should be short routines that cope with whatever triggered the interrupt, then return to the interrupted, lower-priority task as rapidly as possible.

An *interrupt vector* holding the handler's address provides the link between an interrupt source and its handler. In Intel 80*x*86 systems, the interrupt vector's address is simply the Interrupt ID multiplied by four: `INT 08h` uses the vector at address 0020 (hex, naturally). The collection of all 256 (decimal) vectors occupies 1024 bytes of storage and is referred to as the Interrupt Vector Table or, on '286 and higher CPUs, the Interrupt Descriptor Table.

Although *everyone* knows that the IVT starts with the `INT 00h` vector at address 0000:0000, it turns out that the `LIDT` instruction (Load Interrupt Descriptor Table) can set the table's starting address and maximum length, even in real mode. The power-on default, of course, matches the 0000:0000 and 1024 bytes found in the earliest 8086 CPU. While operating systems may find reason to relocate or resize the IVT, ordinary programs have little need of such shenanigans.

Your interrupt handler code must obey several rules, which we'll explore throughout this chapter, as it executes. The basic rule is simple: the interrupted program should resume execution as though the handler never got control, apart from the time required to complete the handler. Any changes the handler makes must occur to hardware or variables it controls, without affecting the bystanders.

Assuming you've written a good handler, the final piece of the puzzle involves returning from the interrupt handler. Exactly how you do that depends on which type of interrupt your code handles.

For all external and software interrupts, the CPU pushes the `FLAGS` register, the `CS` register, and the `IP` register onto the stack, with the stacked CS:IP indicating the address of the instruction that *would* have been executed had the interrupt *not* occurred. If an external interrupt arrived through the **`INTR`** pin, the CPU then clears the Interrupt Flag to suppress all further external interrupts. Although **`NMI`** interrupts do not actually clear `IF`, the CPU ignores the **`INTR`** pin until the **`NMI`** handler ends with an `IRET` instruction.

Exception interrupts, triggered by the CPU's internal hardware, come in three flavors: faults, aborts, and traps. You must spend some time with the references to understand the differences; you'll see only the first two in real mode, unless you're exceedingly unlucky. For our purposes, faults and aborts push the address of the *current* instruction, the one that failed. Conversely, traps push the address of the *next* instruction, just like external and software interrupts.

Although you can, in principle, fix up the condition that caused a fault and successfully re-execute the failed instruction, this can be difficult in actual practice. A trap, on the other hand, is over and done with by the time your handler gets control, meaning that you must continue with the next instruction, if possible.

With that in mind, the `IRET` instruction at the end of each handler restores the CPU's `FLAGS`, `CS`, and `IP` registers from the stack, precisely what we want regardless of the interrupt's cause. Restoring the `FLAGS` register from the stack restores the `IF` bit, which, if this was an external interrupt, re-enables further external interrupts.

To summarize: when the CPU detects an interrupt, it pushes the `FLAGS`, `CS`, and (usually incremented) `IP` registers on the stack, computes the interrupt vector address from the Interrupt ID (which may be supplied by external hardware, internal hardwiring, or the instruction), fetches the starting address of the interrupt handler from the IVT, and transfers control to it.

When the handler finishes its work, it executes an `IRET` instruction that restores the registers from the stack. The CPU picks up where it left off.

Piece of cake, yes?

The Rest of the Story

The Intel 8259 Programmable Interrupt Controller holds the key to understanding PC interrupts. As with any Intel peripheral bearing the term "Programmable" in its name, the 8259 is a maze of modes, options, control bits, and gotchas. I'll concentrate on how it behaves in a PC and leave the rest for an evening of data book spelunking. Note that, despite the obvious acronym, an Intel 8259 PIC has no relation to a PIC microcontroller made by Microchip Technology, Inc.

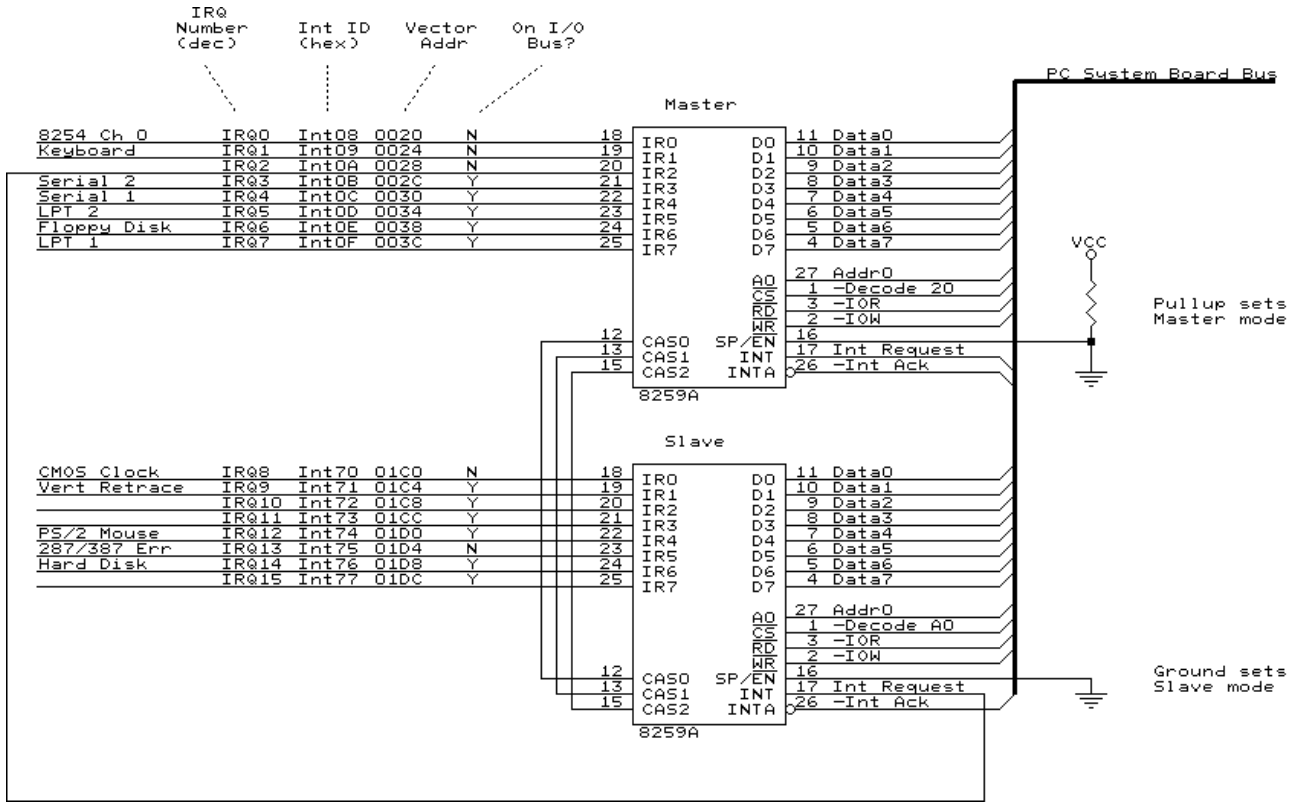
As shown in Schematic 1, the PC system board uses a pair of 8259s in tandem to provide 15 external interrupts. Of course, progress has long since subsumed the 8259 chips into an LSI package (along with all the other CPU support circuitry), but the PC Compatibility Barnacles dictate how that hardware must behave.

Word of warning: if you're contemplating doing anything at all out of the ordinary, remember that the 8259 data sheet does *not* necessarily apply to the LSI chips inside your PC. You should have the real specs on the actual LSI marvel you're using, as modes or functions not used in "normal" PC applications may not work quite correctly. Give them a go, but don't be surprised at unexpected results.

Indeed, even if you have the appropriate specs, don't be surprised if the hardware doesn't quite match the documentation. Often, the folks who design the hardware don't talk to the documentation writers and *neither* discuss matters with the testers who verify everything.

Schematic 1

The system board circuitry devoted to interrupts boils down to a pair of 8259 Programmable Interrupt Controller chips or their LSI equivalents. This diagram summarizes the 8259 Interrupt Request (IRQ) numbers, the CPU Interrupt (INT) numbers, and the vector addresses for each external interrupt. Because the slave 8259 cascades through the master 8259's IRQ 2 pin, IRQ 8 through IRQ 15 have higher priorities than IRQ 3 through IRQ 7. The BIOS redirects IRQ 9 to the IRQ 2 vector maintain compatibility with older PCs; the pin that was IRQ 2 on PCs is labelled IRQ 9 on ATs.



Each 8259 is an I/O device with two internal addresses selected by the **A0** address line. Unlike the 82C54 circuitry in the previous chapter, you cannot use 16-bit I/O cycles with the 8259, because it expects separate I/O operations at each port. The master 8259 resides at addresses 0020 and 0021, with the slave at 00A0 and 00A1.

For what it's worth, the terms *master* and *slave* seem to have fallen out of favor lately. I must continue to use them here, because that's how Intel worded the 8259 data sheet. *Primary* and *secondary* may be more, ah, PC compatible, but you won't find them in the data books and references.

The BIOS initializes the 8259s for normal PC interrupt operation, which should suffice for most embedded PC code. Figure 1 shows an approximate diagram of an 8259 after the BIOS finishes its setup, minus all the control logic and special cases.

Each 8259 has eight Interrupt Request inputs called **IRQ 0** through **IRQ 7**, with the slave's **IRQ** inputs called **IRQ 8** through **IRQ 15** on the PC. By convention, the 8259's **IRQ** inputs bear decimal numbers. These **IRQ** numbers are simply labels with no mystical significance. As you'll see later, though, the slave's **IRQ** numbers are particularly inauspicious.

A rising edge on any **IRQ** input constitutes an interrupt request, so the 8259 turns on the corresponding Interrupt Request Register bit. You can force the 8259 to ignore specific combinations of **IRQ** inputs by turning on the appropriate Interrupt Mask Register bits. A 1 bit in the IMR *disables* the corresponding **IRQ**.

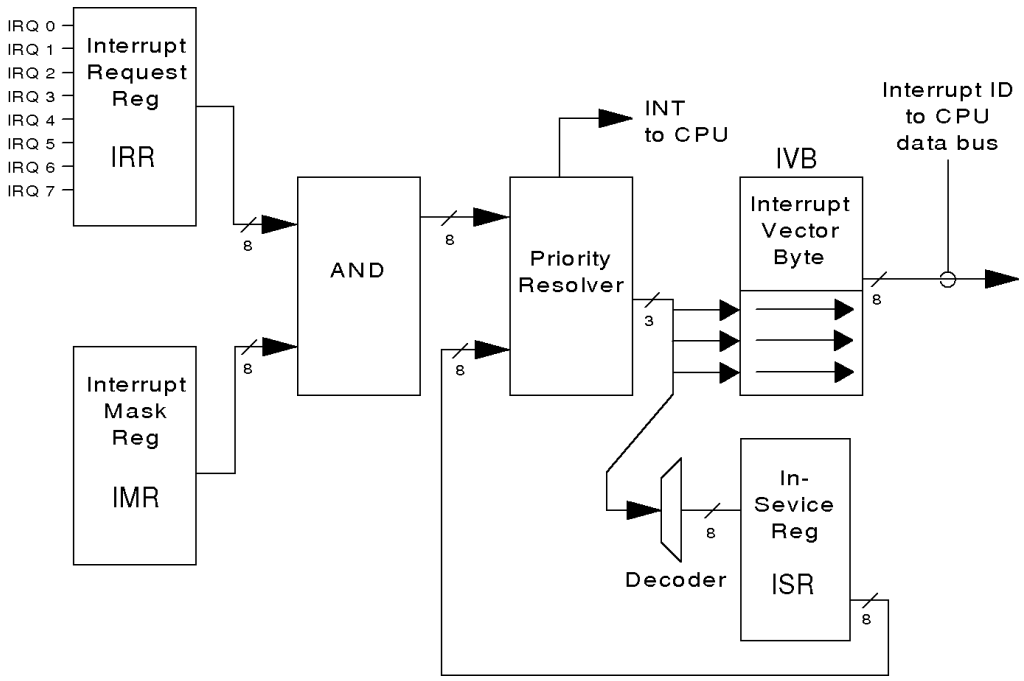
Assuming IMR does not mask the **IRQ**, the 8259 then activates its **INT** output, which raises the CPU's **INTR** input. As described above, that triggers a hardware interrupt if the CPU's Interrupt Flag is set. A 1 bit in `IF` *enables* the interrupt. The CPU blips the **INTA** once to tell the 8259 to resolve its highest priority interrupt.

If two or more IRR bits are 1 simultaneously, the 8259 figures out which has the highest priority and saves the rest for later. The priority rules can be complex and may change on the fly if you reprogram the 8259, but, in normal PC operation, the rule reads *lowest numbered* **IRQ** *wins*.

A considerable amount of time may elapse between an IRR bit going active and the CPU's response. If additional IRR bits become active in the meantime, the 8259 will recognize the highest priority interrupt at the time of the first **INTA** pulse from the CPU. The other IRR bits remain active and will cause additional interrupts as they become the highest priority inputs in turn.

In any event, the winning IRR bit turns on the corresponding In-Service Register (also an ISR, not to be confounded with the Interrupt Service Routine described

Figure 1
Although the 8259 Programmable Interrupt Controller has a bewildering variety of modes and settings, this diagram shows roughly how it works in normal PC operation. A rising edge on an IRQ line sets an IRR bit. If the corresponding IMR bit doesn't mask it, the priority resolver decides if the new IRR has a higher priority than any of the bits currently set in the ISR. If so, it activates the 8259's INT output. When the CPU acknowledges the interrupt, the 8259 combines the IVB with the IRQ number and sends an Interrupt ID to the CPU, then turns on the appropriate ISR bit. At the end of the interrupt handler, an EOI command from the CPU resets the ISR and IRR bits. If any other IRR bits remain active, the entire process starts over again.



above) bit to indicate that the CPU has acknowledged the interrupt request. Several ISR bits can be active at any one time, all but one indicating that its lower-priority interrupt handler has been interrupted by a higher-priority interrupt.

So far, so good?

The CPU blips the **INTA** line once more to tell the 8259 to put the Interrupt ID on the data bus. This bus activity is *not* a standard I/O operation, because the **-IOR** and **-IOW** lines remain inactivate. The **INTA** and **INTR** control lines do not appear

on the ISA bus, which means that you cannot put an 8259 on an I/O board and produce more vectored interrupts for your own use, no matter how delightful that prospect may seem or how much it would simplify your embedded life.

The one-byte Interrupt ID seen by the CPU has two fields: five high-order bits from the 8259's Interrupt Vector Byte register and three low-order bits identifying the current ISR bit. The BIOS loads a different value into each 8259's IVB register during the power-up sequence: the master gets 0x08 and the slave gets 0x70.

Once the CPU reads the Interrupt ID, it proceeds as I described earlier: pushes registers, turns off `IF`, converts the ID to a RAM address, fetches the corresponding interrupt vector, and starts the interrupt handler. These operations occur in lockstep sequence, without any further interruption.

Meanwhile, the 8259 can accept a new interrupt on any **`IRQ`** without an active IRR bit. If an **`IRQ`** goes active, the 8259 compares its priority to the highest ISR bit and recognizes only higher priority **`IRQ`**s by turning on the **`INT`** output. As before, the CPU responds to or ignores its **`INTR`** input depending on the Interrupt Flag's state.

The interrupt handler routine must write an `EOI` (End Of Interrupt) command to the 8259 to clear the ISR bit and re-enable any lower-priority interrupts. The BIOS sets the 8259 up so that what's called a `Nonspecific EOI` will reset the highest-priority ISR bit. You can also issue a `Specific EOI` to reset a different bit, but this isn't usually desirable.

The `Nonspecific EOI` command is 0x20. Yes, folks, that's identical to both the master 8259's base I/O address *and* the first address of its interrupt vectors in RAM. You can see why named constants are such a good idea… you cannot tell what the number 0x20 is, represents, or will do, just by staring at it.

Interrupts on the slave 8259 work slightly differently. As you can see from Schematic 1, the slave's **`INT`** output connects to the master's **`IRQ 2`** input and the two chips have different **`SP/-EN`** input settings. The slave **`IRQ`** activates its **`INT`** output after resolving its own **`IRQ`** priorities. That signal triggers the master's **`IRQ 2`** and forces the usual priority resolution. When the CPU responds to the master's **`INT`**, however, the *slave* provides the Interrupt ID.

In this case, two ISR bits become active: one in the slave 8259's ISR that identifies the actual interrupt and ISR bit 2 in the master 8259 that indicates an active slave 8259. The interrupt handler (forgive me for not calling it an ISR) for each slave **`IRQ`** must send an `EOI` to each 8259 to clear both ISR bits.

Collision Alarm!

The default setting for the master 8259's Interrupt Vector Byte register may be the second-worst idea in the whole PC kingdom. I had long believed it represented a classic case of what happens when you ignore what's printed in the data books, but the actual story has a surprising twist.

As you saw earlier, events inside the CPU generate exception interrupts. Intel now reserves Interrupt IDs 00h through 1Fh for those exceptions, but the 8086 and 8088 did not use all 32 IDs (nor do the 80486, the Pentium, and the Pentium Pro, for that matter). The PC BIOS code used several of those reserved interrupts for its own software functions and hardware interrupts. Inevitably, when subsequent Intel CPUs triggered exceptions using interrupts already used by the BIOS, the two functions collided with a loud crash.

In late 1996, Dave Bradley presented a history of the IBM PC at an IEEE session in Research Triangle Park, NC. When he mentioned that he wrote the Original PC BIOS, I asked him why he used those particular BIOS software interrupt numbers. He just rolled his eyes. Obviously, he'd gone through this before.

He said it was a simple case of Hobson's Choice. Microsoft had already laid claim to all the interrupt vectors upward from 20h for their own software. In late 1980, Intel had not yet reserved *any* of the unused interrupts below 20h; that came later, when Intel codified their 80186/80188 designs. So, Dave simply put the BIOS functions and hardware interrupts in the only vacant spots shown by his data books.

Then, in 1982, Intel published those 80186/80188 design manuals and laid claim to those interrupts. Allowing for the usual publication leadtimes, they had no way of knowing that the IBM PC was about to become the tail wagging their dog.

As usual, there's enough blame to go around. However, the moral of this story remains the same: don't tread on reserved ground! If you *know* of a restriction, you *must* assume somebody will eventually use it… even if your design lacks the staying power of the IBM PC architecture.

In any event, Figure 2 summarizes the conflicts between the CPU, BIOS and hardware interrupt IDs. Many of the CPU exceptions occur only in protected mode, where the real-mode BIOS becomes irrelevant. BIOS and DOS hardcode the remainder in a thick layer of PC Compatibility Barnacles.

You can avoid the conflicts at `INT 08h` through `INT 0Fh` by writing a different Interrupt Vector Byte into the master 8259. Although only protected-mode operating systems require this, we embedded systems types may find it a handy

Figure 2
Intel's documentation now reserves Interrupt IDs between 00h and 1Fh for CPU exceptions, but the IBM PC BIOS and interrupt hardware got there first. This table summarizes the collisions. CPU exceptions marked with an asterisk occur only in protected mode, making real-mode BIOS conflicts nearly irrelevant. You can change the 8259's Interrupt IDs by writing a new value into the Interrupt Vector Byte register.

Int ID	Intel CPU Exception	BIOS Function	PC Hardware IRQ	
00	Divide-By-Zero Error			
01	Step/Debug			
02	NMI pin			
03	Breakpoint			
04	INTO (Overflow)			
05	Bound Check	Print Screen		
06	Invalid Opcode			
07	x87 Not Available			
08	Double Exception		IRQ 0	RTC
09	287 Segment Overrun		IRQ 1	Keyboard
0A	Invalid TSS *		IRQ 2 / IRQ 9	Video
0B	Segment not present *		IRQ 3	COM2
0C	Stack fault		IRQ 4	COM1
0D	General protection		IRQ 5	LPT2
0E	Page fault *		IRQ 6	Diskette
0F	Reserved		IRQ 7	LPT1
10	x87 Error (-ERR pin)	Video functions		
11	Alignment Check (486+) *	System info functions		
12	Machine Check (Pentium+)	Get memory size		
13	Reserved	Disk I/O functions		
14	Reserved	Serial I/O functions		
15	Reserved	System functions		
16	Reserved	Keyboard functions		
17	Reserved	Printer functions		
18	Reserved	Cassette BASIC (!)		
19	Reserved	Bootstrap loader		
1A	Reserved	Time Functions		
1B	Reserved	Ctrl-Break handler		
1C	Reserved	System timer tick		
1D	Reserved	Video parameter tables		
1E	Reserved	Diskette param tables		
1F	Reserved	Pointer to 8x8 graphic font		

trick. The standard alternate seems to be `INT 50h`; if you're going to be nonstandard, you may as well pick the standard method.

I'll leave those machinations as exercises for you, as we already have entirely too many details in this chapter. You should certainly peruse the data books before trying anything fancy. But… you knew that already, didn't you?

While you're doing that, notice that the slave 8259 **`IRQ`** inputs, **`IRQ 8`** through **`IRQ_15`**, use the same digits as the master 8259's Interrupt IDs. Many references list them both in decimal, adding to the confusion: it sure looks like **`IRQ 15`** (decimal) ought to match up with `Int 15` (decimal), doesn't it? At least now you have a fighting chance of seeing why **`IRQ 15`** isn't related to `Int 0fh`…

Three Timers Ticking

Having stunned the subject, let's back up and run it over. The `TimeTest.C` demo program produces three external interrupts at known rates using the Firmware Development Board's 82C54 timer, then reports on the handler response times. As before, I wrote the code in Micro-C. You can load it on your target system using either the diskette boot routine or `MON86`'s serial `HEX` transfer command.

The program puts all three 82C54 timers into Mode 2, which produces a short blip when the counter reaches zero. I picked a 5 ms period for the timers to simplify scope sync: 200 traces per second make it easy on the eyes and allow plenty of time for the handlers to finish up before the next interrupt.

Timers 0, 1, and 2 drive **`IRQ 5`**, **`IRQ 10`**, and **`IRQ 15`**, respectively. The master 8259 drives **`IRQ 5`**. Both **`IRQ 10`** and **`IRQ 15`** have higher priorities, because they come through **`IRQ 2`** by way of the slave 8259.

The first order of business is writing an interrupt handler. Although, for reasons that will soon become evident, I favor assembly language over C for this task, Listing 1 shows a perfectly serviceable Timer 0 handler written in Micro-C. It resembles a standard C function, except for the `INTERRUPT` macro appearing before the function name.

You should remember that C functions and their inline assembly code may change nearly anything, including the CPU registers, before returning to their caller. This is obviously inappropriate behavior for an interrupt handler that must return control to the interrupted program with all registers intact. Although the Micro-C compiler saves and restores `BP`, we must handle the remaining registers by ourselves.

Listing 1
Interrupt handlers are generally written in assembly language, but you can get away with C in some cases. This handler turns on a parallel printer port bit, records the current timer value in an array, sends an EOI to the 8259, and turns off the parallel port bit. The INTERRUPT macro on the first line is the key to using a standard Micro-C function as an interrupt handler. See Listing 2 for the macro definition.

```
/*--------------------------------------------------------------------------- */
/* Timer 0 hardware interrupt handler                                         */
/* Assumes interrupt on master controller                                     */

INTERRUPT(Timer0Handler) HandlerT0() {

   outp(SYNC_ADDR,inp(SYNC_ADDR) | 0x01);

   if (!(Response[0][RESP_SET]++)) {
      Response[0][RESP_NOW] = ReadTimer(0,I8254_BASE);/* fetch times       */
   }

   outp(I8259A,NS_EOI);

   outp(SYNC_ADDR,inp(SYNC_ADDR) & ~0x01);

   return;

}
```

Dave Dunfield suggested the `INTERRUPT` macro shown in Listing 2. It `PUSH`es all the CPU registers and the `?temp` variable used by the Micro-C compiler and runtime routines, then `CALL`s the C function, which can be as rude as it chooses. When the function returns, the `INTERRUPT` macro `POP`s the saved values off the stack and performs the obligatory `IRET` instruction after restoring all the registers.

Different compilers turn standard C functions into interrupt handlers using different methods, but they all boil down to essentially the same thing: save and restore the CPU state around the C code. Check your compiler's manual for the details, then write a few testcases to verify that you understand what's going on.

It should be obvious that hardware interrupt handlers cannot have any parameters, nor may they return results in the CPU's registers. By definition, all the CPU registers and the stack will be in an unknown state (as far as the C code knows) when the interrupt occurs, so the handler must restore them to the same condition as it exits. Those restrictions leave no place for parameters nor return values!

With that in mind, `HandlerT0` in Listing 1 turns on bit 0 in the parallel printer port, reads Timer 0 and records the value in a global array, sends the all-important `EOI` to the 8259, and shuts the printer port bit off. Triggering your scope on

Listing 2
A C function invoked by a hardware interrupt must not change anything used by the mainline C code. This Micro-C macro saves all the CPU registers and the compiler's ?temp scratch variable, calls the interrupt handler, then restores everything before returning.

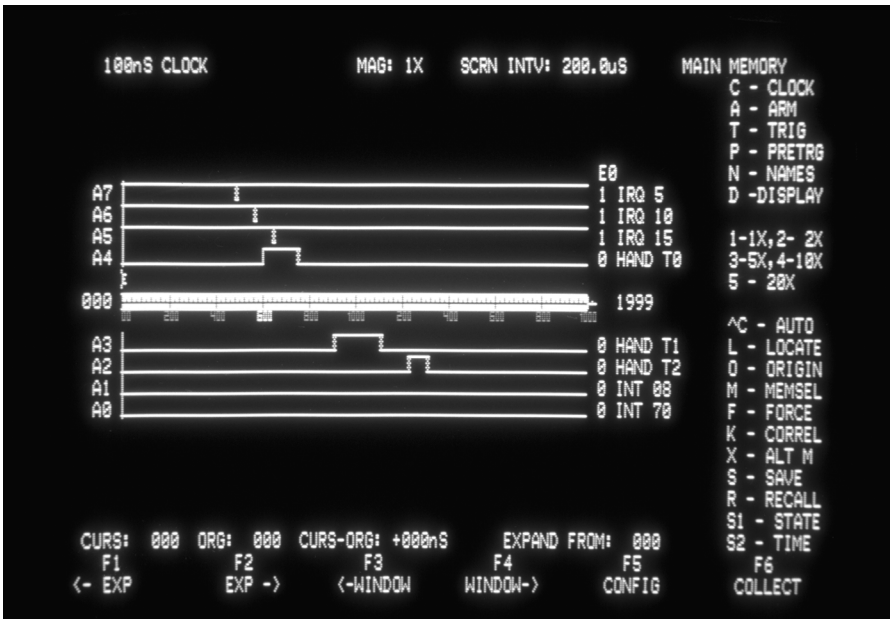
```
#define INT_PROLOGUE 30                    /* size of prologue code        */

#define INT_ENTRY(fn) (&fn-INT_PROLOGUE)   /* start of prologue            */

#define _SPACE_
#define INTERRUPT(fn) asm { \
fn PUSH  AX \
   PUSH  BX \
   PUSH  CX \
   PUSH  DX \
   PUSH  SI \
   PUSH  DI \
   PUSH  ES \
   PUSH  DS \
   PUSH  CS \
   POP   DS \
   MOV   AX,?temp \
   PUSH  AX \
   CALL  fn+INT_PROLOGUE \
   POP   AX \
   MOV   _SPACE_?temp,AX \
   POP   DS \
   POP   ES \
   POP   DI \
   POP   SI \
   POP   DX \
   POP   CX \
   POP   BX \
   POP   AX \
   IRET \
}

#undef _SPACE_
```

IRQ 5 and observing bit 0 on the printer port should give you a good indication of how much time your CPU takes to respond to an interrupt.

The **IRQ 10** and **IRQ 15** handlers are similar, with a few wrinkles I'll discuss in a moment. Photo 1 shows the three **IRQ** pulses triggering their interrupt handlers. Notice that the **IRQ 5** handler runs to completion, even though the other two handlers have higher priorities (remember: **IRQ 10** and **IRQ 15** trigger **IRQ 2** on the master 8259, which outranks the master's **IRQ 5**). `HandlerT0` runs with interrupts off, because the CPU disables other external interrupts when it accepts the first one: those other handlers don't stand a chance.

Listing 3 shows the **IRQ 10** handler. Compiling with `NEST_1` defined causes the code to send `EOI`s that enable interrupts immediately after setting the parallel port bit. Photo 2 shows the result: `HandlerT2` now interrupts `HandlerT1.` Even

Photo 1
The IRQ signals shown in the top three traces trigger three interrupt handlers that report their activity through the target system's parallel printer port. This logic analyzer record shows that HandlerT0 will run to completion, even with higher priority interrupts pending, because the interrupt handler code does not set the CPU's Interrupt Flag bit.



though **IRQ 15** has a lower priority than **IRQ 10**, the 8259 thinks that the IRQ 10 handler has finished when it receives the corresponding EOI.

One implication of this situation: a second **IRQ 10** hardware interrupt would start another copy of HandlerT1! As far as the 8259 is concerned, HandlerT1 is history when it sent an EOI command to shut off its In-Service Register bit. In our case, the handler has finished long before the next IRQ 10 pulse, but it's something to bear in mind if you have fast interrupts and slow handlers.

Photo 3 shows a somewhat more complex event: HandlerT1 can be interrupted by more than one other interrupt handler. In this case, the BIOS timer tick on INT 70h (**IRQ 8**) and HandlerT2 get into the act. The timer tick has a higher priority than **IRQ 15**, so even though **IRQ 15** remains pending, the **IRQ 8** handler runs first.

Listing 3
If NEST_1 is defined, this interrupt handler will send an EOI command to the 8259 interrupt controllers and turn on the CPU's Interrupt Flag bit. That allows other interrupt handlers to gain control in the middle of this code. If the 82C54 timer were to produce another IRQ before this code returns, the CPU would push its registers again and start executing the handler from the top, which is generally not what you expect.

```
/*-------------------------------------------------------------------------*/
/* Timer 1 hardware interrupt handler                                      */
/* Assumes interrupt on secondary controller                               */

INTERRUPT(Timer1Handler) HandlerT1() {

   outp(SYNC_ADDR,inp(SYNC_ADDR) | 0x02);

#ifdef NEST_1

   outp(I8259B,NS_EOI);               /* tell slave we are done           */
   outp(I8259A,NS_EOI);               /* tell master we are done          */
   enable();                          /* and allow other interrupts       */

#endif

   if (!(Response[1][RESP_SET]++)) {
      Response[1][RESP_NOW] = ReadTimer(1,I8254_BASE);/* fetch times       */
   }

#ifndef NEST_1

   outp(I8259B,NS_EOI);               /* tell slave we are done           */
   outp(I8259A,NS_EOI);               /* tell master we are done          */

#endif

   outp(SYNC_ADDR,inp(SYNC_ADDR) & ~0x02);

   return;
}
```

The source code includes the routines I wrapped around the BIOS interrupt handlers to activate parallel port bits for those pictures. This trick comes in handy, even for application programmers. Well, application programmers who care about their code's realtime performance… firmware folks, just like you and me.

Digital Readout

Even if you don't have a scope or logic analyzer, you can still experiment with interrupt handlers using the code from this chapter. Each handler reads back the current value of its 82C54 timer channel and stores it in a global array. Once per second, the mainline code calculates the minimum, running average, and maximum values of those times for each channel, then sends the results to the host system through the serial port, where it appears as shown in Figure 3.

Photo 2
A different version of HandlerT1 sends EOIs to the 8259s and sets the CPU IF bit as soon as it gains control. This record shows HandlerT2 running while HandlerT1 is suspended, even though HandlerT2 has a lower priority. As soon as the 8259 receives an EOI, it resets the highest priority ISR bit and generates a new INT output based on whatever IRR bits remain active, which can result in precisely the situation shown here.

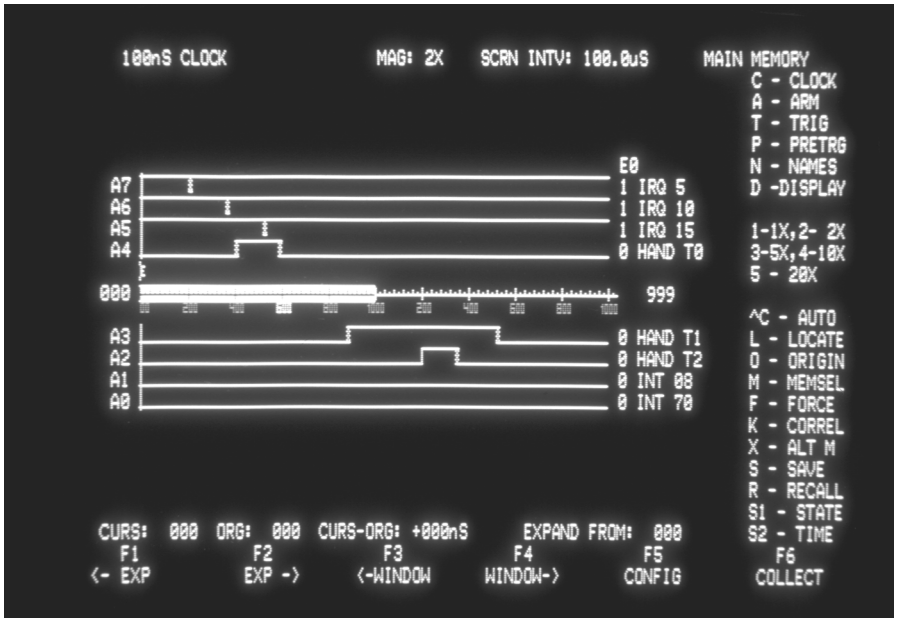
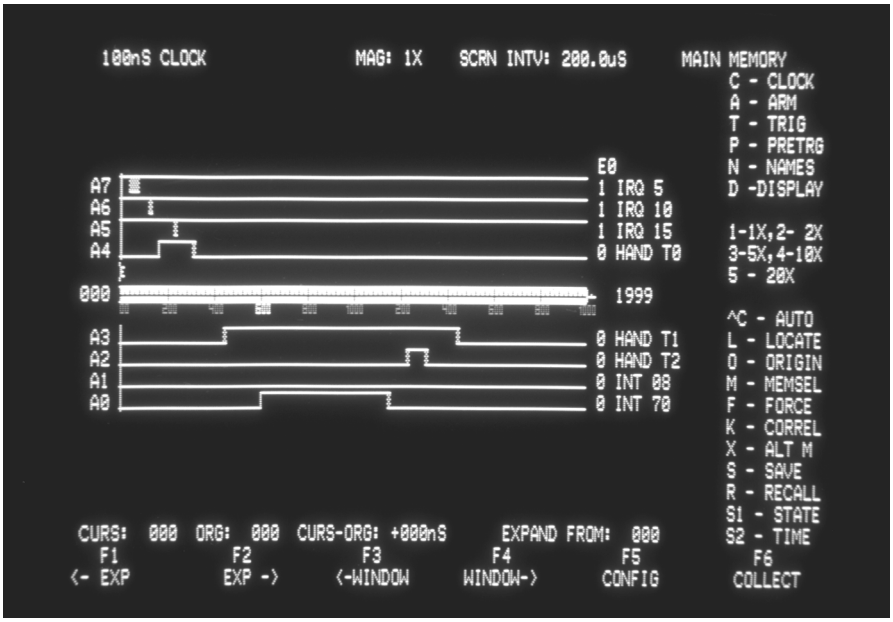


Figure 3
The interrupt handlers record the elapsed time between the IRQ signal and the 82C54 command required to latch the timer. The main program summarizes those values, showing you how the different interrupt handlers respond. Displaying this response time table on your host PC requires a terminal emulator capable of handling ANSI cursor control strings, which is usually a matter of selecting the right option from a dialog box.

```
Interrupt Handler Timing Exerciser
Embedded PC's ISA Bus Chapter 5 -- Ed Nisley
Timers are not synchronized
<<< some text omitted in this listing >>>
Interrupt response times in 139 ns ticks:
Timer Current Minimum Average Maximum
  0       102      96      97     545
  1       114     108     108     597
  2        40      35      35     524
Max stack used: 00C8
```

Photo 3
This photo shows both the Real-Time Clock and HandlerT2 gaining control during HandlerT1's execution. The RTC interrupt on INT 70 runs first because it has a higher priority than HandlerT2 and does not re-enable interrupts until it ends.



Recall that 82C54 timers running in Mode 2 count down to zero, generate an output blip (which we wired to an I/O bus **IRQ** line), then automatically reload their maximum count value and continue to tick. The difference between the maximum and current timer values equals the number of ticks since the reload, precisely the elapsed time since the **IRQ** line went active.

Thus, the minimum and maximum values shown in Figure 3 give you an estimate of how fast your handler can possibly respond to an interrupt and how long it may take under less-than-ideal conditions. These figures change as the program continues to run and the handlers interrupt each other in different patterns.

You can easily see that `HandlerT2`, triggered by **IRQ 15**, runs considerably faster than `HandlerT0` on **IRQ 5**. Listing 4 shows the reason: assembly language code living inside a standard Micro-C function.

Listing 4
This interrupt handler uses assembly language to reduce the overhead caused by saving and restoring the CPU state. The result is a much faster handler that is also much harder to understand, a familiar tradeoff in embedded systems work.

```
/*-------------------------------------------------------------------------*/
/* Timer 2 hardware interrupt handler                                      */
/* Assumes interrupt on secondary controller                               */
/* CAUTION -- all the constants are hardcoded in here...                   */
/*  ... and the registers are saved manually                               */
/* Micro-C inserts PUSH BP and MOV BP,SP before the first line of code     */

HandlerT2() {
asm {
              PUSH      AX                  save bystanders
              PUSH      DX
*
              MOV       AL,#$80             latch Timer 2
              MOV       DX,#$030E
              OUT       DX,AL
*
              MOV       DX,#$0378           flag startup
              IN        AL,DX
              OR        AL,#$04
              OUT       DX,AL
*
              MOV       AX,CS               set up data segment addressing
              MOV       DS,AX
*
              PUSH      SI                  save more bystanders
              PUSH      DS
*
              MOV       DX,#$030            read the latched LSB
              IN        AL,DX
              JMP       <Punt4
Punt4         NOP
              MOV       AH,AL
              IN        AL,DX                ... and the MSB
              XCHG      AH,AL               rearrange them
              MOV       DX,AX               save for later
*
              MOV       SI,#Response+(2*7*2)         aim at our slice of Response
*
              MOV       AX,4[SI]            fetch _SET element
              AND       AX,AX               previous entry processed?
              JNZ       Done                nonzero says skip this one
*
              MOV       2[SI],DX            stash it away
              INC       ASWORD(4[SI])       account for this sample
*
Done          MOV       AL,#$20             send the EOIs
              OUT       $20,AL               ... to master
              OUT       $A0,AL               ... and slave
*
              MOV       DX,#$0378           flag shutdown
              IN        AL,DX
              AND       AL,#$FB
              OUT       DX,AL
*
```

Listing continued on next page

Listing continued from previous page

```
            POP     DS              restore bystanders
            POP     SI
            POP     DX
            POP     AX
            POP     BP              saved at function entry
            IRET                    preempt normal return
}
}
```

Unlike the other two handlers, this one need not save and restore all the CPU registers, because it knows precisely which ones it will use. Therefore, it gets started faster, runs faster, and exits faster. In round numbers, it runs in a third of the competition's time… while becoming at least three times harder to write and understand. You may choose which end of that tradeoff you prefer.

The main line code monitors the total stack used by the routines, which changes while the program runs. This will be particularly noticeable with nested interrupts, as the varying combinations use an unpredictable amount of stack space.

Unlike my blown cerebral stack, it's essentially impossible to blow the stack in this program, because Micro-C's setup code puts the stack at the far end of the 64 KB code and data segment. It's worthwhile to check the program's stack requirements occasionally, but 64 KB marks a big change (and welcome relief) from an 8031 microcontroller's cramped quarters. For that matter, how much of a program could you write with only 128 bytes of RAM?

Although I used a few other tricks in the source code, this overview should get you started. I heartily recommend spending some time with interrupt handlers to discover what your system can do… and what it *can't* do, no matter how good your firmware skills may be.

Speaking of skills, here's a scary story for you...

A Cautionary Tale

Before you start writing firmware, you *must* read the hardware data sheets carefully. Generally, I do a lot of reading before starting a project, but once in a while, well… This tale shows that, verily, hell hath no fury like that of an unjustified assumption.

Once Upon A Time, back in 1988, I wrote MC-Net, a control program that connected 8052-class microcontrollers and PCs into a networked system. You could write data collection firmware on the microcontrollers, store the results on

the PC's disks, monitor and operate the microcontrollers from the PC, and so on and so forth.

To summarize the details, MC-Net used an RS-485 serial link running at 19.2 kb/s between (up to) 32 nodes. A `Monitor` program, running on a PC AT, served as a remote console for all the 8052-BASIC nodes and provide an overall network debugging display. I described the code in my *Firmware Furnace* column in *Circuit Cellar INK* magazine Issues 10 through 12. All in all, a neat project.

Because the network's design point specified an 8 MHz IBM AT with an 8250 UART, handling a 1920 bytes/second network posed some interesting challenges. A stock AT runs at about 1 MIPS and can execute about 500 instructions between each incoming byte. If you get distracted for a millisecond or so while doing something else, you will certainly lose data, because the 8250 has no FIFO buffers.

Each transmitted byte generates two interrupts, because the RS-485 network echoes outgoing data back to the receiver. The first interrupt occurs when the transmitter buffer goes empty and the second, very shortly thereafter, blinks on when the receiver buffer fills with same character. The elapsed time varies, but it can be as little as one stop bit time, about 50 µs at 19.2 kb/s.

Because the two interrupts occur so close together, I checked for pending interrupts at the end of the handler to eliminate the lengthy interrupt exit and entry overhead when a byte was ready. My scope showed that this worked quite well: most of the time each character produced only a single interrupt as the handler absorbed the second interrupt request.

About a year later, though, some customers reported sporadic problems with network errors that we simply couldn't duplicate. Some of the problems occurred due to cabling errors, some to termination problems, others came from severe noise… but there remained a very small minority of customers with everything set up right and everything going wrong.

The problems seemed more severe on faster machines. Finally, one customer installed MC-Net on his new 66 MHz 486DX2 and reported that it failed in a matter of minutes. *Ah-ha!* The bug *must* be related to CPU speed, because, in this case, we'd carefully eliminated everything else.

I set up a test network on my then-new 33 MHz '386SX, activated the trace outputs built into all my code, hitched up the logic analyzer, and waited to see what happened. After a long wait, the outputs reported that the TSR had jammed in an absolutely *impossible* state.

Although my system wasn't as fast as the latest '486 CPU, the logic analyzer showed that each outbound character generally produced two interrupts. The CPU now ran fast enough that the interrupt handler exited before the second interrupt occurred. Progress had eliminated the need for my interrupt polling trick, although the code was still in place for slower CPUs. However, very, *very* rarely, the second interrupt, the one caused by the receiver buffer, produced a suspiciously long blip.

I modified the TSR code to produce unique trace outputs for each possible interrupt source and discovered that the abnormally long interrupts were caused by a change in the Modem Status Register (MSR). That was certainly peculiar, as the TSR code never enabled MSR interrupts… and the Interrupt ID Register (IIR) should *never* report a disabled interrupt.

Essentially all PCs use National 8250, 16450, or 16550 serial interface UARTs, or a fragment of LSI that works just like them. I pored over the data sheets in search of something I'd missed three years earlier. What could possibly cause an invalid IIR? I assumed that my code was at fault, as genuine hardware problems remain very, very few and very, very far between.

In the 8250 family, you clear the transmitter interrupt flag when you read the IIR or write a new character, but the only way to clear the receiver interrupt is reading the pending byte. You would expect, as I did, that the chip updates the Interrupt ID Register almost immediately. You would be nearly right.

The IIR reports the highest-priority pending interrupt as a number in Bits 1 and 2. Bit 0 presents a summary status bit that, when zero, means "there is at least one interrupt active." My code read the IIR and used it as an index into a decoding table. Only two other interrupts can occur in the MC-Net TSR, but, being a belt-and-suspenders type, my table had all possible entries. That saved my skin!

Upon closer scrutiny, the 16450 data sheet revealed two key timings. The hardware updates the summary bit within 250 ns of reading the IIR for transmitter interrupts. Should the *receiver* cause the interrupt, however, the hardware may take up to 1 μs to reset the summary bit after you read the character. The interrupt request output pin has the same timings, so the IIR bit must be wired to the output driver rather than the actual logic gate on the chip.

As Sherlock puts it, "When you have eliminated the impossible, whatever remains, however improbable, must be the truth."

A sufficiently fast CPU can respond to the interrupt, read the IIR, branch to the receiver handler, read and process the byte, and check the IIR again *before* the

interrupt summary bit changes. Because the receiver interrupt bit clears almost immediately, the IIR contains invalid data.

As you might guess, an all-zero IIR indicates a modem status interrupt.

I checked my references again to see if anyone else knew about this. The only hint appeared in Mark Nelson's *Serial Communications: A C++ Developer's Guide*, published in 1992 (well after I needed it). In the *8250 Oddities* section, he states:

> One annoying bug found in both the original National Semiconductor chips as well as some clone chips is the false modem status interrupt. The IIR can report a modem status interrupt when none has occurred. This could easily lead to trouble with the ISR code.

Indeed!

The serial port in your system may be just one corner of an LSI chip, but the PC Compatibility Barnacles dictate exactly how it must work. In this case, the Barnacles required my new silicon to precisely duplicate the same old bug.

There being no good fixes for this situation, I used a time-honored kludge: a delay loop. The code measures the CPU speed when it installs the TSR and sets up a delay loop that occupies at least a microsecond. After each receiver interrupt, the code stalls for long enough to ensure that the IIR interrupt summary flag becomes valid before testing it again. Not pretty, but it worked OK… and so did the TSR.

This story has a twofold moral: RTFM (Read The *Fine* Manual) first, then build trace outputs into your code to show you what's going on in actual real time.

But… you knew that already, didn't you?

Right?

Release Notes

I've recoded several of the support routines in assembler and moved them into the `FirmDev.ASM` library file. Use **`SLIB`** to update `Tiny.LIB` on your system.

The program files for this chapter include several batch files that create the executable files. You'll surely need to tweak them for your system, but that's all part of the learning experience. Check the `ReadMe.txt` file in this chapter's subdirectory for more details.