8 Ticks, Pops, and Restarts

Traditional embedded systems start up at the flip of a switch. Even the PC's built-in ROM BIOS lights up immediately. However, as we saw in the previous chapter, loading code from diskette can take quite a while and may conflict with your watchdog timer. There's no quick-and-dirty cure for that, but converting your code into a BIOS extension can help.

In this chapter, I'll explore BIOS extensions in more detail with a set of routines that capture interrupts, support the Firmware Development Board's power failure detection hardware, and record information in nonvolatile storage. You can use either the EEPROM or the battery backed RAM circuitry, as long as the firmware can enable and disable the memory chip's **-WE** line.

The Key to the Code

Schematic 1 shows the new hardware you'll need: a pushbutton switch with a pullup resistor driving bit 9 of the input port at 031C. This may be barely worth warming your soldering iron, but, every now and then, we need an easy one.

Holding that button can become awkward at times, particularly when you're also maneuvering a scope probe or two, so I rewired the front panel keyboard lock switch in parallel with the button. Because we have yet to use the keyboard, I figured the keyboard lock might be superfluous.

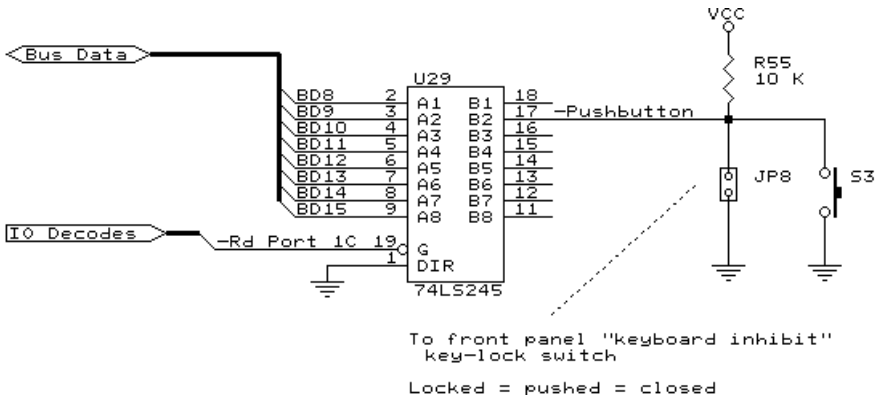
Incidentally, should you ever come up against a "locked" PC clone, just whip out your Swiss Army knife's Phillips blade, unscrew the clone's case, yank (or slice) the lock switch wires, and fire that sucker up. I trust I'm not compromising the security of what was once the Free World by letting that trick out of the bag. The Original IBM PC AT had a lock that disabled the keyboard *and* secured the metal cover to prevent just such an assault.

So much for the hardware. Now, on to the code!

Extension Essentials

The Original PC BIOS didn't permit any extensions, which led to some truly remarkable kludges as each vendor devised different and mutually incompatible ways to glue new functions into old PCs. The method we'll use dates back to a PC BIOS revision slightly before the XT. That means, for all intents and purposes, every IBM PC now handles BIOS extensions the same way.

Schematic 1
This simple hardware, a button and a resistor, allows you to skip over a malfunctioning BIOS extension during a boot and prevent a system crash. The supporting circuitry appeared in Chapter 7 as part of the battery backed RAM and watchdog controls.



The part of FDBExt.ASM shown in Listing 1 sets up the 55 AA signature, length, and checksum bytes required by the BIOS extension scan. Recall that the checksum byte in the source code must be zero, because our diskette boot loader computes the actual checksum as it copies the extension into the Firmware Development Board.

The code at BootEntry forms an escape hatch I suggest you build into all your extensions, at least while debugging them. With the external button pressed (or the key lock switch ON) while booting, FDBExt simply updates the LEDs and returns to the BIOS. This can save your bacon when your new extension crashes the BIOS boot sequence. Trust me. It can happen to you, too.

With the switch released or the lock switch OFF, the code shown in Listing 2 makes the whole software development process I'm using in this chapter work correctly. As Steve Ciarcia often puts it, "Let me explain…"

The Case of the Missing PSP

FDBExt, written in Borland's Turbo Assembler TINY memory model, produces an ordinary COM file. As far as TASM and TLINK can tell, the program will run under DOS with all the usual DOS assumptions and restrictions. Of course, if we actually *do* run it under DOS, it won't work very well at all.

Listing 1
This code fragment resides in the battery backed RAM at C800:0000 on the Firmware Development Board. The BIOS passes control to the instruction at offset 0003 (just after the length byte) during the power-on sequence. This code first checks the pushbutton switch input bit; if the button is pressed, the code immediately returns to the BIOS.

```
; This does not start at offset 0100, so be careful about data accesses!
; Actual execution is at absolute address C800:0003...

            CODESEG
            STARTUPCODE

            DB      055h                ; signature
            DB      0AAh
            DB      2                   ; length in units of 512 bytes

            JMP     SHORT BootEntry     ; force two-byte jump

            DB      00h                 ; loader sets this value

;--- constants that must be stored within the checksummed region

RevCode     DW      0001h               ; current revision level

;--- if pushbutton is down, exit without doing much

BootEntry:
            MOV     DX,STAT_ADDR
            IN      AX,DX
            TEST    AX,PUSHBUTTON
            JNZ     SHORT Continue      ; nonzero means not pushed

            MOV     AX,NOT 0101h        ; show -- to track our path
            MOV     DX,LED_ADDR
            OUT     DX,AX

            RETF                        ; return to normal BIOS boot
Continue:
```

As you know by now, `COM` files contain an exact binary image of the program's code and data. They date back to the days of CP/M and 8080 CPUs, when 64 KB of RAM was all even a big spender could have. The CP/M operating system reserved the first 256 bytes of RAM to control the 8080's reset and interrupt vectors and provide some entry points. Therefore, it loaded your `COM` programs at absolute RAM address 0100.

MS-DOS adopted much the same memory layout, except that 64 KB of RAM suddenly seemed not quite so much after all. A `COM` file fit neatly into one 64 KB segment atop the reserved 256 bytes, which, still filled with operating system stuff, became known as the Program Segment Prefix. DOS handles `EXE` files somewhat differently, their PSPs live in a different segment, and we'll get to them later, but,

Listing 2
With the switch open, the next step adjusts the segment registers. The RETF instruction loads the new values into CS and IP from the stack. Note that this code uses the FS and GS segment registers found in '386 and higher CPUs and will not run on earlier CPUs.

```
;--- adjust CS and DS to simulate the normal COM situation
;    we need both code and data starting at offset 0100 rather than 0000
;    C800:0000 is also C7F0:0100, so we just subtract 0010 from the segments
;    Storing this CS in the vectors allows normal access after an interrupt

             MOV       AX,NOT 0100h       ; show single - here
             MOV       DX,LED_ADDR
             OUT       DX,AX

             MOV       AX,CS
             SUB       AX,0010h           ; adds 100 to offsets in segment
             PUSH      AX
             PUSH      OFFSET BootStart
             RETF                         ; set CS and IP to new values

BootStart:
             MOV       AX,0057Eh          ; show r0 on LEDs to mark entry
             CALL      ShowBits

             PUSH      DS                 ; preserve seg regs
             PUSH      ES
             PUSH      FS
             PUSH      GS

             MOV       AX,CS              ; set up DS to match CS
             MOV       DS,AX
             MOV       ES,AX

             MOV       AX,0               ; FS points to 0000:xxxx
             MOV       FS,AX

             MOV       AX,0040h           ; GS points to 0040:xxxx
             MOV       GS,AX

             CALL      OpenRAM            ; enable writes

             INC       [ES:ResetCtr]      ; count this reset
```

for now, the key point remains that `COM` files start at 0100 for simple, historical, largely arbitrary reasons.

Although all of the code and data addresses within a `COM` file assume that it's loaded at offset 0100, the actual disk file does *not* include those first 256 bytes. The instruction at offset 0100 within the program's code segment resides at offset 0000 relative to the start of the *file*. DOS must set up the segment registers so that the offsets become correct within the segment where the program will be loaded.

The diskette boot loader in Chapter 1 simulates this process. It loads your program from diskette into RAM at address 1000:0100 and places nothing at all in the first

256 bytes of the segment at 1000:0000. Although there's no PSP, that trick let us use standard COM files without invoking a specialized linker. As long as our program doesn't expect anything in the PSP, having a blank (or random) one is OK.

The PC's BIOS, on the other hand, knows nothing of this. When it finds our Firmware Development Board extension, it passes control to the branch instruction with CS:IP set to C800:0003. Because our extension will set interrupt vectors as well as change data, it must somehow adjust the segment registers on the fly.

The solution involves simple subtraction, because a given physical address can be represented by many different segment and offset values. The CPU hardware shifts the segment register left by four bits, adds the offset, and takes the result as the physical address. At least that's the case in real mode, which is all we require now; protected mode programming is *entirely* different.

In our situation, the branch instruction at C800:0003 resides at physical address C8003. We can also call that address C7F0:0103, because C7F00 + 00103 = C8003. Thus, if we reload the segment registers with C7F0, rather than C800, and change IP from 0003 to 0103, all our offsets become correct and we can use COM files for BIOS extensions.

The easiest way to reload both CS and IP is by yank them from the stack with a RETF (Far Return) instruction. Listing 2 shows the trick in all its glory… not particularly impressive, hmmm? Just try to figure out what it does without some commentary, though. Notice that the PUSH instruction operates on the value of OFFSET BootStart as defined in the COM file, not BootStart's absolute offset within segment C800.

COM programs generally assume that all four of the CS, DS, ES, and SS registers have the same value, but CS and SS hold the essential values for our code. I load DS and ES from the adjusted CS value. SS cannot point into the nonvolatile memory, because that RAM has hardware write protection.

Fortunately, as long as we only PUSH, POP, CALL, and RET from the stack, whatever values the BIOS put in SS and SP will work fine. I have not investigated how deep the default BIOS stack may be. Should your program need lots of stack space for some peculiar reason, you should certainly create a stack somewhere else.

Although I didn't find this written down anywhere, the BIOS in my system requires that you restore at least DS and ES before the extension returns. As usual, the final RETF restores CS from the stack. I save and restore all the segment registers, even though the requirements surely depend on which BIOS you're using.

FDBExt also marks a departure from the code you've seen so far: notice that I'm now using the FS and GS segment registers that appear only in '386 and higher CPUs. As a result, this code will *not* run on 8088 or 80286 systems. I don't include any tests for the CPU type, as I assume we're all responsible folks around here. Don't try this on your old clunker PC just to see what happens… it won't work!

Yes, we could rewrite FDBExt to work on any 80*x*86 CPU, but it's long past time to start using hardware that's been around since 1985. OK?

Capturing Interrupts

The remainder of FDBExt's initialization code captures the BIOS timer and Non-Maskable Interrupt vectors. You've seen similar routines in Chapters 5 and 7, so refer back to those source files if you need a refresher.

Listing 3 shows the timer interrupt handler. The Firmware Development Board's RAM has its write protection hardware active, forcing each handler to enable writes before updating its variables. The CS segment stored in the interrupt vector equals the DS value set up in Listing 2, allowing the INC instruction to reach TickCtr using CS without saving, loading, using, and restoring DS.

Because the RAM write enable bit shares the same port as the watchdog timer bit, I made the OpenRAM and CloseRAM routines toggle the watchdog on each BIOS

Listing 3
The BIOS extension captures the BIOS timer interrupt to count the ticks since the most recent reset. The value of CS stored in the interrupt vector allows access to the extension's variables in nonvolatile memory. Because this code fragment was assembled in '386 mode, the INC instruction increments a 32-bit counter in one shot and the final JMP uses the SMALL keyword to specify that OldTimer holds a seg:off address.

```
PROC      TickHandler

PUSH      AX
PUSH      DX

CALL      OpenRAM             ; enable writes

INC       [CS:TickCtr]        ; TickCtr is 32 bits wide!

CALL      CloseRAM            ; disable writes

POP       DX
POP       AX

JMP       SMALL [CS:OldTimer.DWORD]

ENDP      TickHandler
```

timer tick. Measuring the bit's active time shows that the interrupt handler requires about 30 µs on a 33 MHz '386.

As I mentioned in Chapter 7, it's generally a Bad Idea to toggle a watchdog from a timer interrupt, because the main routine can crash while the timer tick continues running. However, this trick can keep the watchdog at bay while loading a big program from diskette. The mainline code can always capture the timer tick and implement my favored method after it starts running.

Assuming it gets that far, of course! If something goes wrong during the disk boot or program setup process, we've just defeated the entire purpose of the watchdog timer. Think about it, then decide just how paranoid you should be. Which is worse: a failure that you don't detect or a system that doesn't start up correctly?

Enabling '386 assembly mode has some interesting side effects. Even though the `TickCtr` variable occupies four bytes, the assembler updates it with one 32-bit `INC` instruction. The `JMP` at the end of the routine chains to the previous interrupt handler as usual, but you must specify `SMALL` to indicate a 16-bit `seg:off` value, rather than a 32-bit `LARGE` offset in the current segment.

I like that sound… even in real mode!

Failing Power

Our BIOS extension responds to power failures by write protecting the RAM and spinning in a safe shutdown loop. While writing this code, though, I uncovered a nasty bug: **`NMI`** glitches. While they shouldn't pose a problem in most systems, it's worth thinking about them if you're using a power monitor.

The Firmware Development Board includes a trimpot that adjusts the voltage on the MAX691's **`PFI`** pin. The correct setting activates **`-PFO`** when the supply voltage falls near the system's lower tolerance limit; say -5% on a ±10% system. The remaining 5% gives you enough time to shut the system down before the voltage goes completely out of tolerance.

In small microcontroller systems, the MAX691 may be the only source of Non-Maskable Interrupts. In our situation, though, many parts of a PC can contribute to the **`NMI`** signal. Our handler must examine the board's power failure status and either chain to the previous **`NMI`** handler if it finds **`-PFO`** high (inactive) or shut down the system when it sees a low **`-PFO`** input.

Here's the problem: if the supply voltage falls slowly enough, a small supply glitch that would normally be well within tolerance can trigger the power failure

comparator and generate a Non-Maskable Interrupt. By the time the CPU responds to the **NMI** and checks the **-PFO** status bit, however, the glitch has long since Gone Away, even though the voltage continues to fall.

You can simulate this by very slowly adjusting the **PFI** trimpot. Tease it until the system shuts down, then leave the trimpot unchanged. The system will probably shut down sporadically every time you reboot it.

With none of the **NMI** sources active, the interrupt handler chain eventually passes control to the default BIOS handler. Guess what? On my system, the default handler disables further **NMI**s caused by the ISA **-IOCHCK** signal. Thus, when the power really *does* fail after the glitch, the **NMI** handler never gets control.

If the MAX691 presents the only **-IOCHCK** interrupts in your system, your interrupt handler can check the status bit in I/O port 61h to verify that the **NMI** actually came from the bus. Because the system board latches that status bit when -**IOCHCK** goes active, the status will not go away even when the glitch vanishes.

However, if you have several I/O boards that can produce -**IOCHCK** interrupts, the situation becomes somewhat messier. Your handler must test the board's status and chain to other handlers if it finds no problem. An **NMI** glitch from your hardware will set the -**IOCHCK** latch before vanishing. Your handler will find nothing wrong with the PC's power, then incorrectly pass control to other handlers that will also find nothing wrong on their boards.

It should be obvious that this a situation that's best avoided.

In Schematic 2 in the previous chapter, C4, a 1 nF capacitor on the **PFI** trimpot's wiper, filters the glitches with a 10 µs time constant that made the trimpot teaseproof. You should, of course, evaluate this trick in your system to verify that it does not delay the interrupt too much during a real power failure.

An alternative approach, described in the MAX691A data sheet, applies hysteresis around the power failure comparator. Because **-PFO** switches low as **PFI** drops, a high value resistor between those two pins will yank **PFI** down and prevent the end of the glitch from restoring **-PFO**. Assuming, of course, that the comparator's propagation time doesn't glitch it the other way!

You can also add a digital latch that preserves the **-PFO** glitch, much as the system board latches **-IOCHCK**. Remember to include hardware that clears the latch on each hardware reset to prevent a hot **NMI**.

In any event, one of the conditional assembly options shown in Listing 4 inserts a timing loop that starts on the first **NMI**. It displays the loop count on the LEDs until the next **NMI**, at which time it locks up the system. You can use ExtTest.BIN to see how this problem looks on your system.

The CPU automatically disables all interrupts within the **NMI** handler, preventing watchdog timer updates during the final lockup loop. The MAX691 times out and resets the system shortly after the second **NMI** occurs.

Listing 4
This NMI handler normally shuts down the system in response to a power failure. If the supply voltage falls very slowly or if you tease the trimpot controlling PFI, you can get a glitch on NMI that vanishes by the time this routine gets control. The code shown here includes an optional section that displays the elapsed time from the first NMI to the next, then locks up the system. If these glitches pose a problem in your system, the code can also lock up in response to any NMI caused by the ISA -IOCHCK input.

```
                PROC    NMIHandler

                PUSH    AX
                PUSH    DX

                CALL    OpenRAM             ; enable writes

                INC     [CS:NMICtr]         ; record this NMI

                CALL    CloseRAM            ; disable writes

                IF      COUNT_NMI           ; show delay?

                MOV     CX,0                ; set up the counter

@@Wait:
                MOV     DX,LED_ADDR         ; show the loop counter in binary
                MOV     AX,CX
                NOT     AX
                OUT     DX,AX

                IF      USE_IOCHCK

                IN      AL,SYS_CTLS         ; look at IOCHCK flag
                TEST    AL,IOCHCK
@@Kaput:        JNZ     SHORT @@Kaput       ; lock up when it goes high

                ELSE

                MOV     DX,STAT_ADDR        ; check power status
                IN      AX,DX
                TEST    AX,PWR_GOOD
@@Kaput:        JZ      SHORT @@Kaput       ; lock up when status goes low

                ENDIF
```

Listing continues on next page

Listing continued from previous page

```
                INC     CX
                JMP     @@Wait


                ELSE

                MOV     AX,1510h            ; show ni (more or less)
                MOV     DX,LED_ADDR
                NOT     AX
                OUT     DX,AX

                IF      USE_IOCHCK

                IN      AL,SYS_CTLS         ; look at IOCHCK flag
                TEST    AL,IOCHCK
                JNZ     SHORT @@Lockup      ; lock up when it goes high

                ELSE

                MOV     DX,STAT_ADDR        ; check power status
                IN      AX,DX
                TEST    AX,PWR_GOOD
                JZ      SHORT @@Lockup      ; zero = power NOT good...

                ENDIF

                ENDIF

                POP     DX
                POP     AX

                JMP     SMALL [CS:OldNMI.DWORD]
@@Lockup:
                MOV     AX,08080h           ; both decimal points
                CALL    ShowBits
@@Stall:        JMP     @@Stall

                ENDP    NMIHandler
```

Resets and the Worst Hack

The Original IBM PC AT's design engineers faced a serious problem. They needed a way to get their shiny new 80286 CPU back into real mode, even though the chip had no way to shut off its Protected Mode Enable bit. The 80826 emerged from hardware reset in real mode, but once the program entered protected mode, the 80286 architecture provided no way back. Their solution stands as a monument to engineering ingenuity.

The AT included an 8042 microcontroller managing a variety of tasks implemented with discrete logic in the Original PC. The designers simply added a command to the 8042's repertoire that toggled the 80286 CPU's **`Reset`** line.

Blam… back to real mode!

However, the BIOS normally clears the system RAM and runs power-on diagnostics immediately after a hardware reset, which was not quite what they wanted. The engineers reserved a byte at address 0F in the Real-Time Clock's battery backed CMOS RAM to indicate the reason for the shutdown.

Before the BIOS gets too far along in its reset sequence, it asks the keyboard controller what caused the reset to occur. If the controller reports that it executed a `Reset` command, as opposed to participating in a power-on or manual reset, the BIOS reads the shutdown reason code from the clock's RAM. If that byte indicates a transition from protected to real mode, the BIOS branches directly back to the mode switch routine.

The only reason *you* think it's a kludge is that *you* didn't design it. It's really a clean, general, and useful way around an otherwise insurmountable hardware limitation. Remember the Consulting Engineer's First Principle: you don't get paid if the system doesn't work.

Intel got the message loud and clear. Starting with the 80386, all their CPUs enter and exit protected mode at the flip of a bit (well, all right, you need a little setup and takedown code on either side of the bit-flipping instruction, too). By that time, however, a considerable body of software used the Officially Approved '286 method. You can even buy hyperthyroid keyboard controllers with special fast-path hardware logic to recognize and speed up the `Reset` command. I kid you not.

The shutdown reason code in the clock's RAM can select one of several different routines after a reset. Most are ill-suited for civilian use, but one may come in handy in certain desperate situations. I'll show how to use it and you figure out when the trick might be appropriate. Fair enough?

If the shutdown reason code is 0A, the BIOS vectors through the pointer stored at address 0040:0067 to the restart code. That code is normally within the BIOS, but you can redirect the BIOS to your own routine. Because the DRAM refresh hardware continues to run during the shutdown, you regain control immediately after a hardware reset with nearly everything intact. Of course, all the CPU registers except `CS:IP` lose their values, so you must consider a few, ah, minor details that I'll leave as an exercise.

Listing 5a shows `FDBExt`'s rudimentary restart routine, which simply increments a counter and sends a second `Reset` command to the keyboard controller. The BIOS clears the shutdown reason code before branching to the routine and the hardware treats the second reset as a complete, normal, power-on reset.

Listing 5a
This code in FDBExt.ASM gains control after the keyboard controller blips the CPU's Reset line. The BIOS checks the shutdown reason code at address 0F in the Real-Time Clock's CMOS RAM; if that value is 0A it vectors through the address stored at 0040:0067, which ExtTest aims at this routine.

```
                PROC    StartupHandler

                CALL    OpenRAM             ; enable writes

                INC     [CS:StartupCtr]     ; record another startup

                CALL    CloseRAM            ; disable writes

                MOV     AX,00580h           ; show r. on the LEDs
                CALL    ShowBits

                MOV     AL,0FEh             ; tell kbd controller
                OUT     KEY_CMD,AL          ;  ... to blip the reset lind
@@Stall:        JMP     @@Stall

                ENDP    StartupHandler
```

Although our restart handler lives in the Firmware Development Board's nonvolatile RAM, because it *must* exist whenever the CPU uses it, FDBExt cannot write its address into the vector at 0040:0067. At least on my systems here, that restart vector changes *after* FDBExt exits, although the shutdown reason code does not. That means FDBExt must put the address in a spot that ExtTest knows about, but somewhere that the BIOS won't wipe out during its startup processing.

Listing 5b shows the code from ExtTest that transfers the vector from our nonvolatile RAM to address 0040:0067 and sets the shutdown reason code in the clock's RAM. Later, in response to a keyboard command, ExtTest executes this instruction to reset the system using the keyboard controller's reset command:

```
outp(0x64,0xFE);
```

The BIOS then executes the code in Listing 5a, goes through a *second* reset with all the normal power-on tests, and reloads ExtTest from diskette. That's all there is to it. Easy, once you know the secret, isn't it?

Now, if anybody asks you about the Worst Hack in PC-dom, you can say you've been there and done that. Tell me if you put it to good use and what you did with the honorary T shirt.

Listing 5b
This code from ExtTest.C loads the vector and sets the shutdown reason code into the Real-Time Clock's CMOS RAM. Note that the BIOS restart vector cannot be not in the interrupt table because neither a hardware nor software interrupt invokes it.

```
printf("Changing startup vector from %04x:%04x to %04x:%04x...\n",
         peekw(0x0040,0x0069),peekw(0x0040,0x0067),
         peekw(EXTSEG,STARTUP_SEG),peekw(EXTSEG,STARTUP_OFF));
   pokew(0x0040,0x0067,peekw(EXTSEG,STARTUP_OFF));
   pokew(0x0040,0x0069,peekw(EXTSEG,STARTUP_SEG));
   putstr(" setting shutdown code...\n");
   disable();
   outp(RTC_ADDR,0x0f);               /* aim at shutdown reason code       */
   outp(RTC_DATA,0x0A);               /*  ... vector through 0040:0067     */
   enable();
   putstr(" done\n");
```

Release Notes

The files for this chapter include the modified LoadExt boot sector loader that stuffs a BIOS extension into the FDB's nonvolatile storage. You also get FDBExt and ExtTest to show you how the whole process works. The ReadMe.txt file and comments in the source code explain how to load and run the code.

You may find that the interactions between the **NMI** hardware, the BIOS power-on resets, and the BIOS extensions on your target system don't behave quite as I've described. If the sample programs deliver strange results, add some trace outputs and monitor the code's progress through the various stages. A few LEDs that go ON in the right order (or don't go ON at all!) can quickly reveal how your system works.

As always, patience and careful sleuthing will give you far more understanding than applying an In-Circuit Emulator…