10 Booting C from ROM

A foolish consistency is the hobgoblin of little minds…

Ralph Waldo Emerson

That may be an aphorism suitable for any occasion. In this case, back in Chapter 6, I opined that writing BIOS extensions in C probably wasn't practical, citing the Firmware Development Board's limited memory address space. Emerson would smile, as I'll now show you how to do just that.

In point of fact, if you need just a smidge of code you may as well use C and be done with it. After all, it doesn't matter if you have 31 KB of C or 3 KB of tightly written, carefully tuned assembler… that chip has 32 KB of space available. Now that you know how to build battery backed RAM, write code that can send diagnostics through serial ports, and understand the general mechanics of BIOS extensions written in assembly language, we can pull it all together.

In this chapter, we'll explore the gory details of turning a Micro-C program into a BIOS extension. The complexities along the way may seem daunting, but the end result will help you get your own firmware working.

At least, for small values of firmware…

Basic BIOS Booting

As we've seen, every PC goes through much the same ritual immediately after a hardware reset. It first checks the hardware, finds and initializes any BIOS extensions, and finally boots a program from disk or diskette. Each BIOS extension must hook an interrupt vector if it wants to be part of the action after the BIOS regains control and continues booting.

The exact system state isn't predictable when your extension gets control, because any previous extension can add features or change the BIOS setup in nearly any way. In a given system, of course, the same thing happens (or *should* happen) during each boot, but you cannot assume that all systems respond the same way. For example, you'd think that the BIOS would set up the serial ports before invoking the extensions, but, at least on one of my systems, that's not the case.

After calling all the extensions, the BIOS finishes its own initialization and issues an `INT 19h`. Under normal circumstances, that interrupt vector points back into the BIOS code to a bootstrap loader routine responsible for booting from either

diskette or a hard disk. Any BIOS extension, however, can hook `INT 19h` and regain control just before BIOS accesses the disk. That extension may check a serial number, verify a password, or skip the disk boot entirely by booting from ROM.

If the original `INT 19h` BIOS routine eventually gets control in a system with no bootable disks, it will issue `INT 18h` after failing to read the disks. In the Original IBM PC, that interrupt fired up the built-in Cassette BASIC interpreter (remember Cassette BASIC?), but other manufacturers don't have rights to that IBM proprietary code. Most of them simply display a message and await a three finger salute on the Ctrl-Alt-Del keys. Your BIOS extension can hook `INT 18h` to regain control after the disk boot fails, allowing you to start one routine from disk or run another from the extension ROM without booting from disk.

According to the references, the system will be completely ready for action when the BIOS invokes the `INT 19h` and, if needed, `INT 18h` interrupts. Your extension can, therefore, take control of a perfectly functional PC without handling any of the grubby setup work. The sample code for this chapter has a "your code goes here" note at the appropriate spot so you can complexicate it as required.

For what it's worth, Cassette BASIC lives on. My ancient Model 80 (nigh on to a decade old now) proudly displayed its Cassette BASIC screen when I disconnected its disk drive controller to track down the problem I mentioned in Chapter 9. The IBM Tech Reference manual admits that Cassette BASIC might not be too useful, perhaps because Model 80 systems lack a cassette port…

But, well, BASIC is still there!

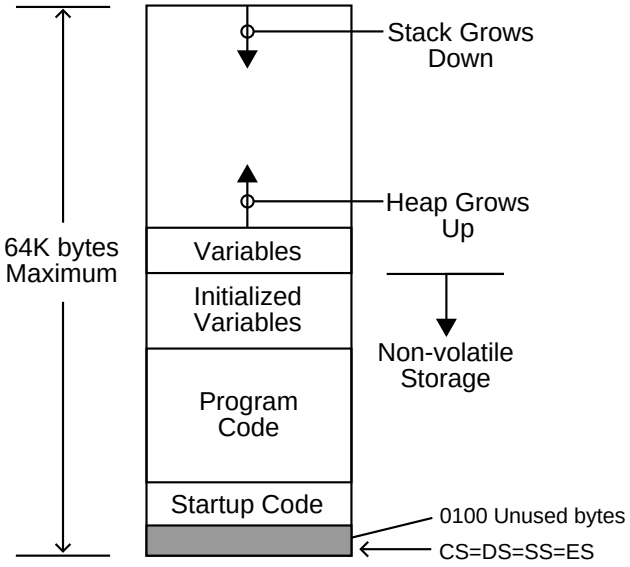
Modeling Memory

Thus far, we have used Micro-C's `Tiny` memory model for our embedded programs. Unfortunately, `Tiny` model won't suffice for BIOS extension code. The reason, as with most things PC-ish, involves the segmented memory inherent in *x*86 real mode programming.

`Tiny` memory model puts all of the C program's code and data into a single 64 KB segment, as shown in Figure 1. When the program begins, all of the CPU's segment registers contain the paragraph address of that segment. An assembly language function or inline code within a C function can refer to memory outside that segment by reloading the segment registers, but it must restore them before returning control to the C code.

The program's startup code begins at offset 0100, followed by the compiled C code and library functions. All of the initialized data, including strings and "constant"

Figure 1
Micro-C's Tiny memory model puts all of the program's addressable storage in a single segment that may be up to 64 KB long. The program code and initialized variables must either reside in nonvolatile storage or be copied from diskette by a loader program. The variables, heap, and stack must be in RAM, for obvious reasons. All of the CPU's segment registers point to the start of the segment.

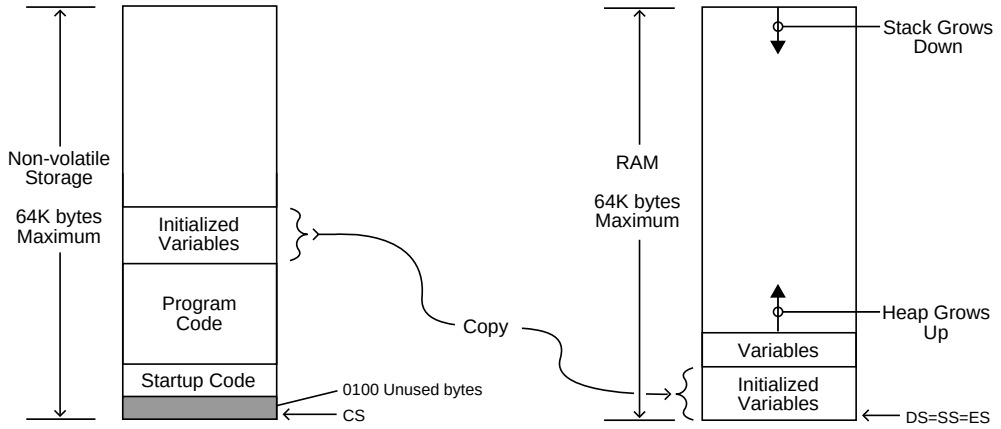


variables, come next. Uninitialized variables follow, with the heap beginning just after the last variable. The stack grows downward from the end of the segment, allowing the heap and stack together to use whatever space remains beyond the program and data.

Recall that the first 256 bytes of the segment remain as a legacy of the DOS `COM` program structure. While this format does let us produce and manipulate the programs with ordinary DOS utilities and compilers, it wastes those 256 bytes. If you have written the program loader to set up the registers properly, as we have, then you can tweak Micro-C's startup code to begin at any offset you'd like, because it need not produce DOS compatible files.

A BIOS extension with a combined code and data segment runs into trouble, because the FDB's battery backed RAM circuitry includes write protection. Storing variables in write protected RAM would be bad enough, but running a CPU with a

Figure 2
Micro-C's Small memory model allows up to 64 KB of program code and initialized variables, which, as with Tiny model, must be in nonvolatile storage addressed by the CS register. The startup code or loader copies the initialized variables into a separate RAM segment addressed by the DS, SS, and ES registers. After that, the main() program may use them during execution without any special attention.



write protected stack simply won't work. The assembler code in Chapter 8 controlled the RAM **-WE** circuitry around each data access, a completely impractical process for a C program. And, of course, the C code must have enough stack space in ordinary RAM for each function's automatically allocated variables.

The most straightforward solution uses Micro-C's `Small` memory model, which puts up to 64 KB of program code and the starting values of any initialized variables in one segment. A separate data segment, also up to 64 KB, holds the actual initialized variables, uninitialized variables, the heap, and the stack. The code segment can be write protected, while the variables reside in ordinary read-write RAM. Figure 2 shows the `Small` model segment layout.

How this works in our situation should be evident. The code segment can live in the FDB's battery backed RAM, while the data segment resides in system RAM below the 640 KB line. The C startup code, in addition to its other duties, must reserve the program's data segment, copy the initialized variables into it, and load the segment registers before calling the `main()` C routine.

It's a simple matter of firmware…

Incidentally, because the startup code copies the initialized variables from nonvolatile storage into RAM, the C program can treat them as ordinary variables with nonzero starting values. Micro-C's `Tiny` model locks the variables into read-only, nonvolatile storage unless you boot from diskette into RAM, as we have been doing all along. In case you hadn't guessed, variables placed in read-only, nonvolatile storage behave a whole lot like constants!

Micro-C includes a `ROM` memory model that directly supports this type of `Small` model programming by copying the initialized variables from ROM into RAM. All we must do is tweak that code into acting like a BIOS extension.

There are other ways to solve the problem, of course. I picked `Small` model because it worked out quite neatly and exploited an interesting Micro-C feature, but another approach may be more suited for your projects. As always, take what you read here and make your own improvements.

For example, you could duplicate the diskette boot loader's function in the BIOS extension. On each reset, your extension would copy the entire C program, initialized variables and all, from the FDB's battery backed RAM into system RAM, then execute it using `Tiny` memory model just as before. The RAM can (and should) remain write protected during the whole operation. If your code fits into 32 KB or you build a paged RAM interface (*ugh*), this can be a perfectly viable way to make it work.

You might also modify the Firmware Development Board's address decoding circuitry to protect the lower 16 KB of battery backed RAM and allow writes into the upper 16 KB, but it certainly seems a shame to leave the entire 640 KB of system RAM unused, doesn't it? Micro-C's startup code includes support for such split memory segments, which are much more common in minimal 8088 systems than full-fledged embedded PCs. However, if you replace the FDB's discrete-logic address decoding with a PAL, you can take advantage of a small, self contained block of RAM that's not directly available to other PC programs.

Sequenced Startup

The C startup code must accept control from the BIOS, adapt Micro-C's runtime conventions to the BIOS extension entry requirements, run the C program, and then return control to the BIOS after the extension finishes its setup. Coupled with the segment shuffling required by `ROM` model code, the C startup code holds some interesting tricks.

The extension must, of course, begin with the signature required by the BIOS scan, so Listing 1a resembles the code presented in Chapter 8. The diskette boot loader

computes the checksum on the contents of the disk file and the values in the battery backed RAM beyond the end of the program. As before, the checksum byte in the extension header must be zero in both the source code and disk file. After the loader returns, the extension is ready to run on the next boot.

The extension bails out without further action if it finds the FDB's pushbutton switch closed, thus allowing a normal boot without loading the extension. Recall

Listing 1a
The ordinary Micro-C startup routine for the ROM memory model assumes it has complete control of the system, so you must make several changes to adapt it to use as a BIOS extension. Pressing the pushbutton on the Firmware Development Board disables this extension and allows a normal boot sequence.

```
*-------------
* BIOS Extension header
*
              ORG       $0100               standard COM offset
*
              DB        $55                 0000 signature
              DB        $AA                 0001 ... second byte
              DB        6                   0002 code + init data size
              JMP       <BootEntry          0003 jump around checksum
              DB        $00                 0005 ... which goes here
*
*-------------
* Now proceed with boot code
*
*--- if pushbutton is down, exit without doing much
*
BootEntry     EQU       *
              MOV       DX,#STAT_ADDR
              IN        AX,DX
              TEST      AX,#PUSHBUTTON
              JNZ       <Continue           nonzero means not pushed
*
              MOV       AX,#~$0101          show -- to track our path
              MOV       DX,#LED_ADDR
              OUT       DX,AX
*
              RETF                          ; return to normal BIOS boot
*
*--- pushbutton is up, so it is OK to continue
*    adjust CS so our offsets are correct
*
Continue      EQU       *
              MOV       AX,#~$0100          show single - here
              MOV       DX,#LED_ADDR
              OUT       DX,AX
*
              MOV       AX,CS
              SUB       AX,#$0010           add 0100 to offsets!
              PUSH      AX                  simulate a FAR CALL
              MOV       AX,#LdRegs
              PUSH      AX
              RETF                          ;  ... to load the new values
LdRegs        EQU       *
```

Listing 1b

Because the Micro-C extension code resides in write protected, battery backed RAM, the data segment must use system RAM to let the program's variables work normally. This section of the startup routine reserves 32 KB just below the infamous 640 KB barrier by adjusting the BIOS Ram Size word at 0040:0013. It stores a pointer to that segment in the battery backed RAM to allow the INT 19h handler to find its data.

```
*--- locate the data segment at the top 32K of contiguous RAM
*    the BIOS RAM size value at 40:13 has units of 1K bytes
*
             MOV       AX,#$057E          show r0 on LEDs to mark entry
             CALL      ShowBits
*
             PUSH      DS                 save BIOS seg regs
             PUSH      ES                  ... on the BIOS stack
*
             MOV       AX,#$0040          fetch memory size
             MOV       ES,AX
             MOV       BX,#$0013
             SUB       ES:[BX],#>$0020    reserve 32K (1K byte units)
             MOV       AX,ES:[BX]         convert RAM size to seg reg
             MOV       CL,#6                (10 bits left, 4 bits right)
             SHL       AX,CL
             MOV       ES,AX              target seg for MOVSB and STOSB
*
*--- copy the initialized data to the data segment
*
             MOV       AX,#$0530          ; r1
             CALL      ShowBits
*
             MOV       AX,CS              aim at this segment
             MOV       DS,AX
             MOV       CX,#?Isize         pick up length
             MOV       SI,#?Ivars         start of init data in code seg
             MOV       DI,#0              copied to start of data seg
             JCXZ      ?nocopy            skip if zero (or 64 K) bytes long

             REP
             MOVSB                        from DS:SI to ES:DI

?nocopy      EQU       *
*
*--- clear the uninitialized data while we have the pointers at hand
*    length is the difference between ?heap and ?temp
*
             MOV       AX,#$056D          r2
             CALL      ShowBits
*
             MOV       CX,#?heap+1-?temp  length of uninit data + heap top
             MOV       AL,#0              get a zero (or a test value!)

             REP
             STOSB
*
*--- aim DS at the new data segment
*
             MOV       AX,ES
             MOV       DS,AX
*
```

Listing continues on next page

Listing continued from previous page

```
*--- save the BIOS SS:SP registers
*     ... begin using our stack in the new data segment
*
                MOV       ?BIOS_SP,SP
                MOV       ?BIOS_SS,SS
*
                MOV       SS,AX               load stack pointer
                MOV       SP,#STK_TOP
*
*--- now save pointer to our data in the battery-backed RAM
*
                MOV       AX,#NV_WENABLE      allow writing into RAM
                MOV       DX,#CTLS_ADDR
                OUT       DX,AX
*
                MOV       AX,#NV_SEGMENT      aim at RAM segment
                MOV       ES,AX
                MOV       BX,#STACKTOPOFF      ... stack offset
                MOV       ES:[BX],SP           ... write offset
                MOV       ES:2[BX],DS          ... and data segment
*
                MOV       AX,#0               prohibit writing into RAM
                MOV       DX,#CTLS_ADDR
                OUT       DX,AX
*
                MOV       AX,DS
                MOV       ES,AX               restore ES
```

that we rewired the keyboard lock switch in parallel with that pushbutton to provide the bypass function without opening the case (assuming, of course, that your PC both has a case *and* that it remains closed). If you omit the button test, you have no way to disable a malfunctioning BIOS extension other than popping the RAM chip out of the board. Trust me, it won't look professional.

The next step adjusts the `CS` and `IP` registers to match the C compiler's `ROM`-model assumptions. The FDB's RAM begins at C800:0000, so the BIOS sets the initial `CS:IP` to C800:0003. The simulated `FAR CALL` reloads `CS` with C7F0, then adds 0100 to `IP`, making the startup code's offsets correct. Refer back to Chapter 8 for more details on this trickery.

The code also must reserve some RAM for the data segment. During power-on, the BIOS self test routine records the system RAM size in kilobytes in the word at 0040:0013. Because this value excludes memory above the 640 KB line, even a 128 MB target system reports a RAM size of 0280 hex or 640 decimal. If your system sports an old CGA board and some specialized hardware, its system RAM can extend up to address B800:0000 for 736 KB of contiguous memory. As I mentioned in Chapter 6, that trick won't work with monochrome or VGA/SVGA boards, or in embedded systems that run without the DOS support required for memory manager programs found in desktop PCs.

The most convenient data segment location, at least for our purposes, lies at the top of system memory. The code in Listing 1b subtracts 32 KB from the nominal RAM size. That means any subsequent BIOS extensions or the embedded program booted from diskette will find only 608 KB of RAM. Admittedly, 32 KB far exceeds what we need for this chapter, but you can see the general principle at work.

The next chunk of code copies the initialized variables from their write protected location in the FDB RAM to system RAM. The Micro-C compiler defines a work variable, `?temp`, as the first uninitialized variable. Because the initialized variables begin at offset 0000 in the data segment, the length of the block of initialized variables is equal to the address of `?temp`. This length will be zero when there are no initialized variables, so we must check the `CX` register before starting the `REP MOVSB` operation to avoid copying 64 KB of variables that don't exist.

Up to this point, the startup code used whatever stack the BIOS passed to it in `SS:SP`. The C program must run with its own stack to guarantee enough space for nested function calls and interrupts, and that stack should be in a known location. The code stores the BIOS's `SS:SP` in the `?BIOS_SS` and `?BIOS_SP` variables, then reloads `SS:SP` with a pointer to the end of the reserved RAM segment. Before returning, the code restores `SS:SP` to keep the BIOS happy.

Figure 2 shows that the stack grows downward in the data segment, so the initial stack pointer aims at the top of the segment. Because this extension uses a 32 KB chunk of RAM, its initial SP should be 7FFE to allow an unused word at the end of the stack. If your code requires a 64 KB data segment, you *must* use FFFE, because the initial `SP` must be at least two bytes below the end of the segment (four bytes, in 32-bit protected mode) to avoid the dreaded Stack Fault error trap when the CPU pops the final value from the stack.

The next step may seem peculiar, but, because our BIOS extension might not be either the first or last extension, we cannot hardcode our data segment's location into the startup code. For example, if an earlier extension claimed 10 KB, our 32 KB segment would begin at 9580:0000 rather than 9800:0000 (work it out!). But, obviously, we can't store a pointer *to* the data segment *in* the data segment, as subsequent code wouldn't know where to look for it.

IBM's BIOS architects reserved the interrupt vectors between `INT 60h` and `INT 67h` for user functions. In principle, our startup code could store the pointer to our segment in the `INT 60h` vector. When the extension begins execution, it could fetch the `INT 60h` vector to find its data. In fact, I used this technique in an earlier version of this code.

Unfortunately, while the books say those interrupts are reserved for user functions, the BIOS in some systems clears their contents after initializing the BIOS extensions and before invoking `INT 19h`. That means any data our BIOS extension stores in that spot simply vanishes. Perhaps those BIOS designers felt that no user code could possibly begin running before that point… they certainly never had us in mind, did they?

The only other spot where we can reliably store information is in the battery backed RAM holding our code on the Firmware Development Board. The last few lines in Listing 1b disable the RAM write protection, store the initial `SS:SP`, and enable the write protection. Because the data segment holds both the variables and the stack, that single pointer provides the initial values for both `DS` and `SS`.

The downside of this technique is that something may clobber the RAM during the few instructions that must execute with writing enabled. Also, you must use RAM, rather than ROM or EPROM, to allow on-the-fly updates as the extension executes. Weigh the risk of overwriting the RAM against the benefits of flexible data allocation to see if this technique makes sense in your application.

The remainder of the startup routine is almost anticlimactic, as you can see from Listing 1c. It initializes the heap by writing a zero at `?heap`, then calls `main()`.

Listing 1c

After preparing the variables and segment registers, the startup code initializes the Micro-C heap and calls the main() function. Unlike most embedded C programs, main() must return to allow the BIOS to continue its boot sequence. This listing shows how the code restores the segment registers and returns to the BIOS.

```
*--- now, for the main event...
*
              MOV       AX,#$0579          r3
              CALL      ShowBits
*
              MOV       <?heap,#0          set up empty heap
              CALL      main               execute main program
*
*--- restore regs and return
*
              MOV       SS,?BIOS_SS        aim back at BIOS stack
              MOV       SP,?BIOS_SP
*
              POP       ES                 from BIOS stack
              POP       DS
*
              MOV       AX,#$0505          show rr to indicate the end
              CALL      ShowBits
*
              RETF                         ; and continue with boot
*---
*** Remainder of Micro-C startup code is unchanged
```

When `main()` returns, the startup code restores the BIOS `SS:SP` values, pops the saved registers from the BIOS stack, and returns to the BIOS through the obligatory `FAR RET`.

If your system has a video board, monitor, and keyboard, the BIOS will make them ready before it invokes your BIOS extension. However, it may or may not prepare optional equipment such as the serial and parallel printer ports. On a stock PC, your BIOS extension can display its status on the screen and read ordinary keyboard input. I'm taking a minimalist approach in the sample code, but don't let that discourage you too much.

Many embedded BIOS extensions will control hardware similar to the Firmware Development Board's LCD panel, watchdog, or serial number. When your extension gains control, you can set your hardware up. Don't go overboard; do just the bare minimum and be done with it. You can report error messages on the standard PC screen, or, preferably, on some LED digits, a heartbeat LED, an LCD panel, or any other hardware that you control.

Listing 2 shows a rudimentary `main()` function that saves the `INT 19h` Bootstrap Loader vector and installs a pointer to a customized `INT 19h` handler. Rather than create a separate storage location for the old vector, the code simply tucks it into the `INT 61h` vector. This is a time-honored PC technique that allows you to invoke the old handler through a simple `INT 61h`, rather than a convoluted indirect `FAR CALL` through a pointer in the data segment. If you must restore the registers before calling the old handler, you'll appreciate why this is a Good Thing.

Listing 2
The main() function shown here moves the INT 19h vector to INT 61h so we can use it later, then aims INT 19h at our own interrupt handler. It sends a tracking output to the parallel port, but cannot send a serial message because the BIOS has yet to identify the hardware ports. Because this function is called during the BIOS extension scan, it must exit to allow the PC to continue booting.

```
main() {

   GetVector(0x19,&Int19Seg,&Int19Off);          /* move old handler      */
   SetVector(0x61,Int19Seg,Int19Off);            /*  ... to new interrupt */
   outpw(CTLS_ADDR,NV_WENABLE);                  /* save in NV RAM, also  */
   pokew(NV_SEGMENT,OLDINT19OFF,Int19Off);
   pokew(NV_SEGMENT,OLDINT19SEG,Int19Seg);
   outpw(CTLS_ADDR,0);

   SetVector(0x19,GetCodeSeg(),INT_S_ENTRY(Int19Handler));

   outp(SYNC_ADDR,0x01);

   return;
}
```

It also stores the `INT 19h` vector in the Firmware Development Board's battery backed RAM to protect it from BIOSes that clear the user interrupts. If you *know* your target system's BIOS clears those vectors or your expect your code to run on many different machines, simply don't store your vectors in system RAM. The code we'll see later checks the `INT_61h` vector against the value in the RAM, re-installs it if the BIOS changed it, and tells you what your BIOS did.

Note that, unlike all the other Micro-C programs we've used so far, this `main()` function *must* exit to allow the remaining BIOS initialization to continue. Because system setup will not be complete while your code executes, the C program should do no more than absolutely necessary. In particular, the BIOS probably hasn't loaded the serial port addresses at 0040:0000, which means `main()` can't send a cheerful "Hello, world!" message through the BIOS serial functions.

Getting the Boot

By intercepting `INT 19h`, the C code regains control just before the BIOS tries to boot from the disk or diskette. I covered such interrupt handlers in Chapter 7, but there are enough differences in the `Small` and `ROM` models to warrant another look.

Listing 3

This macro wrapper uses the Firmware Development Board's battery-backed RAM to locate the handler's data segment and stack. The stk argument determines where the handler's stack begins within the stack segment, thus allowing several simultaneously active handlers to use the same macro. The data locations in RAM must match those used by the program's startup code!

```
#define STACKTOPOFF   (32512+0)
#define DATASEGMENT   (32512+2)
#define OLDINT19OFF   (32512+4)
#define OLDINT19SEG   (32512+6)
#define OLDINT18OFF   (32512+8)
#define OLDINT18SEG   (32512+10)

#define INT_S_PROLOG 64                    /* size of prologue code        */

#define INT_S_ENTRY(fn)  (&fn-INT_S_PROLOG)       /* start of prologue     */

#define _SPACE_
#define INT_SMALL(fn,stk) asm {            \
*--- save bystanders on incoming stack     \
fn PUSH     AX                             \
   PUSH     BX                             \
   PUSH     CX                             \
   PUSH     DX                             \
   PUSH     SI                             \
   PUSH     DI                             \
   PUSH     ES                             \
   PUSH     DS                             \
```

Listing continues on next page

Listing continued from previous page

```
*--- fetch ptr from battery-backed RAM to find DS/ES/SS and stack    \
   MOV      BX,#NV_SEGMENT_A                \
   MOV      ES,BX                           \
   MOV      BX,#STACKTOPOFF                 \
   MOV      CX,ES:[BX]                      \
   SUB      CX,#stk                         \
   MOV      AX,SS                           \
   MOV      BX,ES:2[BX]                     \
   MOV      SS,BX                           \
   XCHG     SP,CX                           \
   PUSH     CX                              \
   PUSH     AX                              \
   MOV      DS,BX                           \
   MOV      ES,BX                           \
*--- save our compiler temp on our stack   \
   MOV      AX,?temp                        \
   PUSH     AX                              \
*--- invoke the Micro-C handler            \
   CALL  fn+INT_S_PROLOG                    \
*--- restore compiler temp from our stack  \
   POP      AX                              \
   MOV      _SPACE_?temp,AX                 \
*--- restore pointer to incoming stack     \
   POP      AX                              \
   POP      CX                              \
   MOV      SS,AX                           \
   MOV      SP,CX                           \
   POP      DS                              \
   POP      ES                              \
   POP      DI                              \
   POP      SI                              \
   POP      DX                              \
   POP      CX                              \
   POP      BX                              \
   POP      AX                              \
*--- bypass Micro-C return                 \
   IRET                                     \
}

#undef _SPACE_
```

Listing 3 shows the macro wrapper for a `ROM`-model Micro-C interrupt handler that knows about our battery backed RAM. It saves a few of the caller's registers on the interrupted routine's stack, recovers the handler's `SS:SP` from the RAM and sets up that stack, saves the rest of the incoming registers on the new stack, then invokes the interrupt handler. When the handler returns, the wrapper undoes all that before finishing the process with an `IRET`.

If your code can generate nested interrupts, such as hardware interrupts during a software interrupt, you must place each handler's stack in a different part of the data segment. The second macro argument simplifies this process: the wrapper code subtracts `stk` from the stack pointer saved in RAM. You can easily allocate separate chunks relative to the original stack top with this trick.

I could use the same stack for both INT 19h and INT 18h, because my INT 18h handler executes after the INT 19h handler returns. In general, however, you must be more cautious with your stacks. In particular, make sure you allocate a different stack area for each hardware interrupt handler and reserve enough space for each stack so that they cannot possibly overwrite each other.

Your INT 19h handler can take over the PC or continue booting as you see fit. The handler in Listing 4 hooks INT 18h, displays a message and waits for a serial character from the host PC before continuing. Depending on what you type, it will either invoke the original INT 19h handler or issue an INT 18h directly. If there are no diskettes in the system, the BIOS INT 19h routine passes control to a customized INT 18h handler.

That INT 18h routine displays a message, then enters an endless loop updating a counter on the FDB's LED display. Obviously, you can be a lot more clever than

Listing 4

The BIOS INT 19h handler normally boots a program from disk, but this handler gives you a choice. Depending on the serial input character it will either boot using the BIOS handler (which the startup code moved to INT 61h) or pass control directly to the INT 18h boot failure handler. You can insert an entire application program either here or in the INT 18h handler, depending on how much hardware setup you expect the BIOS to do.

```
INT_SMALL(Int19,0) Int19Handler() {
int Option;

   serinit(9600,1);
   enable();                                        /* allow timer ticks     */
   putch(FORMFEED);
   putstr("Embedded PC's ISA Bus Chapter 10 -- Ed Nisley\n");
   putstr("Micro-C BIOS Extension Demo\n\n");
   putstr("Interrupt 19 (pre-boot) handler\n");
   outp(SYNC_ADDR,0x02);

/*--- show that we have got proper memory addressing                        */
   StkSeg = peekw(NV_SEGMENT,DATASEGMENT);
   StkOff = peekw(NV_SEGMENT,STACKTOPOFF);
   printf(" Pointer in NV RAM to top of stack is %04x:%04x\n",
          StkSeg,StkOff);

/*--- capture Int 18 vector to route control to our handler                 */
   GetVector(0x18,&Int18Seg,&Int18Off);
   printf(" Capturing Int 18 vector, was %04x:%04x\n",
          Int18Seg,Int18Off);
   SetVector(0x18,GetCodeSeg(),INT_S_ENTRY(Int18Handler));
   outpw(CTLS_ADDR,NV_WENABLE);
   pokew(NV_SEGMENT,OLDINT18OFF,Int18Off);
   pokew(NV_SEGMENT,OLDINT18SEG,Int18Seg);
   outpw(CTLS_ADDR,0);
```

Listing continues on next page

Listing continued from previous page

```
/*--- see if Int 61 survived the BIOS boot process                          */
   GetVector(0x61,&Int61Seg,&Int61Off);
   if (Int61Seg != peekw(NV_SEGMENT,OLDINT19SEG)) {
      putstr("*** BIOS overwrote our info in the Int 61 vector!\n");
      printf("  we wrote %04x:%04x\n",
              peekw(NV_SEGMENT,OLDINT19SEG),
              peekw(NV_SEGMENT,OLDINT19OFF));
      printf("  is now %04x:%04x\n",Int61Seg,Int61Off);
      SetVector(0x61,peekw(NV_SEGMENT,OLDINT19SEG),
           peekw(NV_SEGMENT,OLDINT19OFF));
      putstr("  Restored to proper value!\n");
   }

/*--- now run the command loop                                             */
   while (1) {
      putstr(" Press Enter to reboot or Esc to invoke Int 18h: ");
      Option = getch();
      putch('\n');
      switch (Option) {
      case '\n' :
         printf(" invoking old Int 19 through Int 61 -> %04x:%04x...\n",
             Int19Seg,Int19Off);
asm {
         INT $61
}
         break;
      case ESC:
         printf(" invoking our Int 18 handler...\n");
asm {
         INT $18
}
         break;
      }
   }
}
```

that. Remember, however, that this is the last chance you get to affect the system. I don't know what the BIOS will do if the `INT 18h` handler returns, as Cassette BASIC offered no way back to the BIOS boot code.

To summarize, the C startup code and `main()` function cooperate to form a BIOS extension that gets called during the boot sequence. The `main()` code may hook the `INT 19h` and `INT 18h` interrupts to regain control before and after the disk boot. Those routines are software interrupt handlers, rather than BIOS extensions, and should use the C interrupt macro wrapper to save and restore the registers.

Your code can hook other software or hardware interrupt vectors to replace, modify, or extend standard BIOS services. For example, you could redirect calls to the BIOS `INT 10h` video routines to your character LCD panel routines… that panel is a mite cramped, but your embedded applications could use the LCD or a normal video display with no changes. Get the picture?

Release Notes

Because BIOS extensions require a specialized C startup file that isn't useful for normal C code, I created a separate library file called BIOSExt.LIB that includes the startup code module 8086RLXR.ASM. You can modify BIOSExt.LIB to include specialized library files or exclude Micro-C library files as needed.

The macro wrapper shown in Listing 3 is in the firmdev.h file in this chapter's subdirectory. Unlike the Tiny model wrapper, it requires enough setup and specialized tweaks that I didn't want to put it in the Micro-C subdirectory.

See the source code and ReadMe.txt files for more information.

The Micro-C CC86 Command Coordinator cannot specify a library other than the standard ones, so I've been using my MCComp.BAT file to create BIOS extensions. You can simply replace the 8086RLPR.ASM file with (renamed) 8086RLXR.ASM and use ROM model if you find that more convenient.

You must have the FDB's battery backed RAM and write protection circuitry installed to use this code, because it updates the RAM contents during the BIOS boot sequence.

I *still* think 32 KB doesn't provide enough room for a complete PC program, but now you have a framework for small, C language BIOS extensions and diskless programs. Just don't get carried away and smash into that 32 KB limit at, oh, say, 90% of the way through your next project.