2 Chickens and Eggs

Now that we have a way to boot programs into our ISA bus target system, what should the First Program be? A multitasking, multiuser, multimedia, protected mode operating system featuring 32-bit flat memory model programming with no artificial segment limits?

Hardly!

The whole point of this book is providing the detailed knowledge you need in order to write good firmware and build custom hardware for an ISA bus embedded PC. While I suppose you can simply ignore all of the low-level details, I've found that this kind of knowledge is indispensable in ferreting out bugs, taking maximum advantage of the machinery, and generally doing a good job.

The First Program should be a debugger. The Second Program should be a simple C program. Next, we will look at the hardware used in typical ISA bus interfaces. After that… well, all things in due time.

Despite the killer GUI debuggers now in vogue for PC program development, a traditional command line debugger has a lot going for it, at least for simple projects. Those "primitive" debuggers are relatively small, you can drive them from a serial port, and they do what you need to get things started from scratch. If you have ever used the DOS `Debug` program (and who hasn't?) you know what I mean.

But, because our target system doesn't have DOS, we can't use good old **Debug**. Rather than write my own debugger, I'll take advantage of the Dunfield Development Systems 8086 Micro-C Developer's Kit. It includes the Micro-C compiler, an assembler, a pair of debuggers, and a host of other utilities useful for embedded *x*86 program development.

There are, of course, higher-powered ANSI standard C/C++ compilers that can generate code for embedded PCs and, as we'll see, ways to run GUI debuggers through serial ports and network links. Dunfield's programs are straightforward enough that we can see the fundamentals quite clearly. We'll get more complex in successive chapters, but, for now, let's get the basics up and running.

The Bare Essentials

The diskette boot loader I described last in Chapter 1 provides a simple and inexpensive way to load a program into the target system. In the next chapter, we will begin building a Firmware Development Board that will eventually hold some

nonvolatile memory that can make the diskette optional, but the code from this chapter will help us check out that board and the code that runs on it.

Recall that our target system includes an integrated I/O board with a pair of serial ports or a system board with all that built in. Because we are not using the keyboard or display, those two ports provide an ideal way to communicate with the firmware. It seems a shame to let a resource like that go to waste, so I'll use COM1 for the firmware and COM2 for the debugger that helps get the code running.

Dunfield's `HDM86` (which stands for Hardware Debug Monitor) is one of the slickest chunks of code I've seen in a while. It will run on any 80*x*86 CPU and uses no external RAM: *all* of its data stays in the CPU's registers, including the return address for one level of subroutine call. Talk about tight coding!

Although I won't require `HDM86` for our projects, it comes in handy to verify hardware startup functions such as DRAM refresh and memory mapping. Of course, `HDM86` must be in EPROM (or loaded from diskette, as we will do) and you can't download any code (where would it go?), but performing the actual I/O operations "by hand" can often reveal circuit problems.

`HDM86` includes commands to run tight read/write loops that hammer on a specific I/O or memory address. This makes it easy for you to scope out problems, because the hardware strobes provide convenient sync points. Of course, you must reset the system to regain control, but that's why `HDM86` can live in EPROM.

Incidentally, CPU and system board hardware setup is one of the (many) problems we've sidestepped entirely by using a standard ISA bus PC system board as our target system. As far as we are concerned, the BIOS initializes everything and passes control to us through the diskette boot loader. While we can change anything necessary (and we certainly will, as you'll see later on), the essential startup details are a done deal.

Remember, the rules are changing. You give up detailed hardware diagrams to gain PC compatibility and a rock bottom per-unit price. All things in PC-dom are negotiable, so you can get schematics if you're using system boards in production quantities and you can write your own BIOS if you must have absolute control. However, the relentless pace of PC progress means any system you use today will be obsolete in six months, so don't plan on using any tricks that work on just one system. Stay generic and stay alive…

`MON86`, the larger of Dunfield's two 8086 debuggers, occupies a little over 5 KB (*gasp!*) and includes a disassembler, `HEX` file loader, up to eight code breakpoints, and the usual `Debug` style commands. It arrives set up for the PC's keyboard and

video display, but is easy enough to adapt to any standard PC serial port. Dunfield includes a detailed set of comments to help you through the process, which involves changing a few constants and reassembling the code.

The diskette boot loader handles standard COM programs starting at offset 0100, but MON86 assumes its programs begin at 0000. The only change required is to set MON86's initial User PC (Program Counter, which Intel fans know as the Instruction Pointer: IP) to 0100 instead of 0000.

MON86 disassembles code into 8086-level instructions, but this is not a serious impediment during the first few projects because plain old 8086 instructions provide all the functions we need. When instructions found on '386 and higher CPUs will benefit our code, we'll use an appropriate assembler and debugger.

Load and Go

You could burn the modified MON86 into an EPROM, but I put it on a diskette and boot it into the target system's RAM using the loader from Chapter 1. Remember that the whole point of this exercise is building the tools to build the board to hold that EPROM.

Your host system must have a serial port and communications program to talk with MON86 on the target PC. Schematic 1 shows the three possibilities for the null modem adapter between the host and target system serial ports. With standards like this, it's a wonder we get anything working at all. Be careful of commercial null modems, because I've seen some that cross-wire all of the CTS/RTS and DTR/DSR pins together. Hard to believe, but true.

When MON86 starts up on the target system, you should see a display similar to Listing 1 on your host system's comm program. If not, recheck the source in MON86.MAC to verify that you have all the options set correctly.

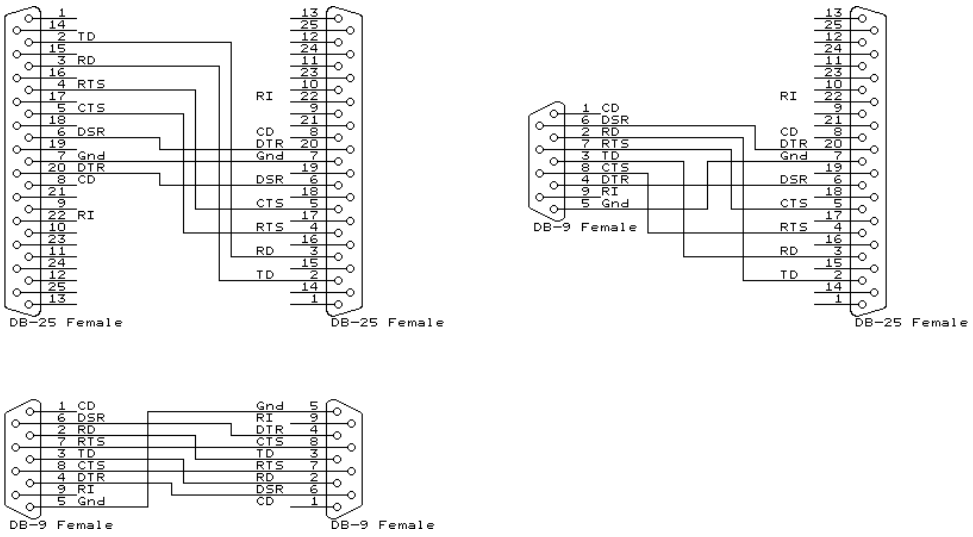
Listing 1

This is what MON86 sends from the target system to the comm program running on your host system. If you get something else (or nothing at all), make sure that both MON86 and your host system comm program are set up correctly... and doublecheck the serial cable! MON86 uses a * prompt rather than the more familiar - or > characters.

```
MON86 Version 1.1
Copyright 1991,1995 Dave Dunfield
All rights reserved.
*
```

Schematic 1

Connecting two PC serial ports together requires a null modem adapter to swap the input and output pins. Use a breakout box or continuity checker to verify the connections inside commercial null-modem adapters, as I've found some that were wired incorrectly. You need not connect the other pins. For example, because the communication program on the host PC will not be dialing or answering a modem, the RI and CD pins are unused.



Although I should not need to mention this, make *sure* you plug the serial cable into the right ports on both PCs. I speak as an expert, having gotten *both* ends wrong on the first crawl under the desk.

Once MON86 boots up, use its **Output** command to write a few values to the parallel printer port at address 0378. The LED and switch circuit shown in Chapter 1 will come in handy to watch the bits toggle. This simple step verifies that MON86 is loaded, the serial ports and your comm program work, and the parallel port is at the address you expect.

The next step involves loading a HEX program with, naturally enough, the **Load** command. Rotate.ASM, shown in Listing 2, has a simple loop that rotates a single bit across the parallel port. It uses a BIOS delay function to make the output visible, so you won't absolutely need a scope to check this code out.

Listing 2
Use MON86 to load Rotate.HEX through the serial port. The LED and switch hardware described in Chapter 1 will show you the results appearing on the parallel printer port. The code shown here should be assembled with Dunfield's ASM86 assembler; the syntax is slightly different than Microsoft's MASM or Borland's TASM, but no more unusual than some other assemblers.

```
                ORG     $100
*
                MOV     DX,#$0378       Select parallel port
                MOV     AL,#$80         Set up the rotating bit
*
ReShow          ROL     AL,1            Rotate it
                OUT     DX,AL           And display it
*
                PUSH    DX              Save our values
                PUSH    AX
*
                MOV     CX,#1           Use BIOS delay function
                MOV     DX,#0            ... to get 64K microseconds
                MOV     AH,#$86
                INT     $15
*
                POP     AX              Recover our values
                POP     DX
*
                JMP     <ReShow
```

Dunfield's assemblers can produce either Intel or Motorola HEX files and M0N86 will accept either flavor without any special attention. However, if you plan to send the files to an EPROM programmer or use them as an input to other programs, be sure you select the appropriate assembler command line switches.

M0N86 does not echo the characters it receives while transferring the HEX data, which means you may have to configure your comm program to *not* wait for the end of each line. If the comm program seems to hang after sending the first line, the two programs are probably locked in this deadly embrace. Cancel the transfer, reset the comm program's options, and try again.

Dunfield's HEXFMT utility will transform Rotate.HEX into Rotate.BIN, which the diskette boot loader will pull directly into RAM. This is the main advantage of changing M0N86's default `User PC`: you can debug programs using M0N86, then boot them directly from diskette with our loader, without making any changes.

The C compiler forms the last piece of the puzzle, as you might expect. It turns out that the effort is almost anticlimactic because Dunfield has done nearly all of the required work already.

Micro-C Startup

This may come as a shock to some C firmware programmers, but there's some startup code that runs immediately after the CPU comes out of a hardware reset, long before your `main()` function gets control. Regardless of whether you program in C on a microcontroller or an *x*86 CPU, you must understand the how, what, and why of that startup code. Only after the startup code runs successfully can your program execute your instructions.

On PCs running a standard operating system, of course, all this is taken care of for you. Now that our target system runs without even DOS, it's time to learn more about the machinery you've been taking for granted all along.

The "bare metal" startup begins when an *x*86 CPU emerges from hardware reset by fetching the first instruction. Each of the 8086, 80286, 80386, 80486, and Pentium CPUs start with slightly different `CS:IP` register values, but the system board maps the resulting physical address to FFFF0 (a.k.a. F000:FFF0) in the ROM BIOS.

As I mentioned earlier, the PC's BIOS handles all the initializations required to get from that first instruction fetch to a system that can boot an operating system or, better yet, your embedded program. If you are writing *x*86 embedded code without the luxury of a BIOS, you must know the details of the target system's hardware so you can provide all the right startup functions.

Several vendors offer "BIOS kits" that reduce the tedium of rolling your own BIOS. Even if you don't use all the standard functions, starting with known-good code can trim months off your schedule. Check the Sources appendix and the ads in your favorite magazines; this is a small but significant niche market.

For our present purpose, however, the BIOS sets up the hardware, reads our boot loader from the diskette, sets up the segment registers, and branches to the loader. Our code reads in the first DOS file, sets up the segment registers, and branches to whatever is at offset 0100. That's where the C startup code must begin.

Listing 3 shows the top part of `8086RLPT.ASM`, Micro-C's `Tiny` memory model startup code. It sets up the segment registers (again), clears the uninitialized variables, and starts the C program with a `CALL` to the `main()` function. If the `main()` function returns, the default Micro-C startup code invokes the `INT $21` Terminate Program DOS function, which is appropriate for DOS-based programs.

I replaced the termination code with a simple `JMP` back to the top, but even that isn't strictly necessary. As you will see later, our embedded C programs never end

Listing 3
The Micro-C startup code for the Tiny memory model loads the segment registers, clears the uninitialized variables, and calls the main() function. I modified the code just after the CALL instruction to restart the program rather than invoke a DOS function we don't have.

```
              ORG $100
*
Restart       EQU       *                 *** restart here on main() exit
              MOV       AX,CS             Get CODE segment
              MOV       SS,AX             Set SS
              MOV       SP,#$FFFE         Initial stack pointer
              MOV       DS,AX             Set DS
              MOV       ES,AX             Set ES
* Zero uninitialized RAM area
              MOV       DI,#?temp         Point to uninitialized RAM
              MOV       CX,#?heap+1-?temp Get size of uninitialized RAM
              XOR       AL,AL             Get a zero
              REP                         Repeat prefix
              STOSB                       Zero memory area
              CALL      main              Execute main program
              JMP       <Restart          *** bypass DOS exit!
* Exit function... Return to MSDOS
exit          XOR       AH,AH             Function 0 - exit
              INT $21                       Exit
```

and that `JMP` should never be executed. Nevertheless, I feel better with a valid instruction instead of a guaranteed trip to the weeds; call me compulsive.

Reloading the segment registers three or four times in quick succession is esthetically displeasing, but of no real consequence because the values are the same. The way I've set things up, each step in the chain is more or less independent of the others, so the new register loads appeared Just In Case the previous program isn't under my control.

Because the diskette boot loader handles only `COM` files, all the code and data must reside in one 64 KB segment. This implies all the segment registers have the same value, which is precisely the definition of the `Tiny` memory model. Micro-C can also produce `Small` model code, which uses separate Code and Data segments, but we will get along quite happily with a single segment until we explore BIOS extensions in Chapter 10.

Console I/O

The Micro-C runtime library includes the usual console I/O functions that connect `getch()` and `putch()` with the outside world. Despite the fact that they're in a file called `SerIO.ASM`, `getch()` delivers characters from the PC keyboard and `putch()` sends them to the video adapter. This makes sense for those embedded

applications that use PC hardware, but our code doesn't yet have the luxury of standard PC I/O. Remmber what's missing from Photo 1 in the previous chapter?

Fortunately, it's easy to substitute the corresponding serial I/O routines. While the BIOS serial functions are not adequate for real life serial I/O, they will suffice for our immediate purposes. Higher performance is, as Steve Ciarcia puts it, "a simple matter of software" that you can work on when the occasion arises.

Listing 4 shows the Firmware Furnace console I/O routines in SerIOFF.ASM. Use the SLIB utility from Micro-C's library utility to replace SerIO.ASM in Tiny.LIB:

```
slib i=tiny.lib a=serioff.asm r=serio.asm
```

Your code must call `serinit()` to specify the bit rate and COM port number. The data rate is limited to 9600 b/s, the maximum allowed by BIOS function `INT 14h, AH=00h`. For speeds up to 19200 b/s, use `AH=04h`. Beyond that, however, you must twiddle the serial port hardware directly.

Although `serinit()` can set up any one of the four standard serial ports, the typical integrated I/O board has two ports, so MON86 uses COM2. You may use the same port for both your C program and the debugger, but the output can be confusing at times.

Finally, we are ready for a C program…

Listing 4

Micro-C's default console I/O routines use the target system's BIOS video and keyboard functions. This code substitutes the corresponding BIOS serial I/O functions, which, while not adequate for heavy-duty use, are good enough to get us started.

```
* Replacement for SerIO.ASM to use serial port
* Embedded PC's ISA Bus -- Ed Nisley
*
*--------------------
* Set up the serial port
*  serinit(bitrate,COMport)
* Assumes 8N1 format
*
serinit       MOV       BP,SP              Get pointer to params on stack
              MOV       BX,DS              set up ES for SCAS
              MOV       ES,BX
              MOV       AX,4[BP]           fetch bit rate (stacked first)
              MOV       CX,#8              Set table length
              MOV       DI,#?bittab        Set table base
              REPNE
              SCASW                        Search for rate
```

Listing continues on next page

Listing continued from previous page

```
                MOV       AL,CL
                NEG       AL                  Convert count to table index
                ADD       AL,#7                (if not found, use last entry)
                MOV       CL,#5               Move bits 2:0 to 7:5
                ROL       AL,CL
                AND       AL,#$E0             clean out low bits
                OR        AL,#$03             Set no parity, 1 stop, 8 data
                MOV       AH,#0               BIOS function number
                MOV       DX,2[BP]            Get COM number (stacked last)
                DEC       DX                  Make COM1 = 0
                MOV       ?COMport,DX         save for later
                INT       $14                 Do the BIOS initialization
* set up modem control outputs to ensure good communications
                MOV       BX,?COMport         get port index
                ADD       BX,BX                ... convert to word index
                MOV       AX,#$0040           aim ES at BIOS data area
                MOV       ES,AX
                MOV       DX,ES:[BX]          fetch port address
                ADD       DX,#$0004           aim at modem controls
                IN        AL,DX               fetch current bits
                OR        AL,#$03             force RTS & DTR high (inverted)
                OUT       DX,AL
                RET
*
*-------------
* Table of BIOS-acceptable bit rates
* The entry index is the BIOS bit rate code value
*
?bittab         DW        110                 0
                DW        150                 1
                DW        300                 2
                DW        600                 3
                DW        1200                4
                DW        2400                5
                DW        4800                6
                DW        9600                7
*
*
* Write a string to the serial port
putstr          MOV         BX,SP             Address stack
                MOV         SI,2[BX]          Get string
?putstr         MOV         AL,[SI]           Get char
                INC         SI                Skip to next
                AND         AL,AL             End of string?
                JZ          ?1                Yes, exit
                CALL        ?putch            Write it
                JMP         <?putstr          Do it all
*
* Write char with NEWLINE<>CR translation
*
putch           MOV         BX,SP             Address stack
                MOV         AX,2[BX]          Get character
?putch          CALL        ?putchr           Write value
                CMP         AL,#$0A           Newline?
                JNZ         ?1                No, skip it
                MOV         AL,#$0D           Add carriage return
                JMP         <?putchr          Write it out
```

Listing continues on next page

Listing continued from previous page

```
*
* Write char with no translations
*
putchr        MOV       BX,SP             Address stack
              MOV       AX,2[BX]          Get character
* Write through IBM/PC BIOS call
?putchr       MOV       DX,?COMport       use our port
              MOV       AH,#$01           Write char to port
              INT       $14               Call BIOS
?1            RET
*
* Hang until a character arrives
* Return value is raw character in AL
*
waitchr       MOV       AH,#$03           Status request
              MOV       DX,?COMport       use our port
              INT       $14               Ask BIOS
              TEST      AH,#$01           Check data ready
              JZ        waitchr           Char is ready
              MOV       AH,#$02           Get character
              MOV       DX,?COMport       use our port
              INT       $14               Ask BIOS
              XOR       AH,AH             clear high byte
              RET
*
* Test for character (No translation)
*
chkchr        MOV       AH,#$03           Status request
              MOV       DX,?COMport       use our port
              INT       $14               Ask BIOS
              TEST      AH,#$01           Check data ready
              MOV       AX,-1             assume no char
              JZ        ?ccr              correct, bail out
              MOV       AH,#$02           Get char
              MOV       DX,?COMport       use our port
              INT       $14               Ask BIOS
              XOR       AH,AH             discard flags
?ccr          RET
*
* Receive char with no translations
*
getchr        CALL      waitchr           Wait for character and get it
              RET
*
* Test for character (with CR<>NEWLINE translation)
*
chkch         MOV       AH,#$03           Status request
              MOV       DX,?COMport       use our port
              INT       $14               Ask BIOS
              TEST      AH,#$01           Check data ready
              MOV       AX,#0             assume nothing
              JZ        ?cch              correct
              MOV       AH,#$02           Get character
              MOV       DX,?COMport       use our port
              INT       $14               Ask BIOS
              XOR       AH,AH             discard flags
?cch          RET
```

Listing continues on next page

Listing continued from previous page

```
*
* Receive char with CR<>NEWLINE translation
*
getch          CALL      waitchr             Wait for char and get it
               XOR       AH,AH               Zero HIGH
               CMP       AL,#$0D             Carriage return
               JNZ       ?1                  No, its OK
               MOV       AL,#$0A             Convert to NEWLINE
               RET
*
* Receive a string: getstr(buffer, size)
*
getstr         MOV       BX,SP               Address stack
               MOV       SI,4[BX]            Get buffer address
               XOR       CX,CX               Zero CX
?2             CALL      getch               Get character
               CMP       AL,#$08             Backspace?
               JZ        ?3                  Yes, handle it
               CMP       AL,#$7F             Delete?
               JZ        ?3                  Yes, handle it
               CMP       AL,#$0A             End of line?
               JZ        ?4                  Yes, handle it
               CMP       CX,2[BX]            Are we at end?
               JAE       ?2                  Yes, don't accept
               MOV       [SI],AL             Write it
               INC       SI                  Advance pointer
               INC       CX                  Advance length
               CALL      ?putch              Write it out
               JMP       <?2                 And proceed
* Delete character from buffer
?3             AND       CX,CX               At beginning?
               JZ        ?2                  Yes, ignore
               MOV       AL,#$08             Backup
               CALL      ?putch              Output
               MOV       AL,#' '             Space over
               CALL      ?putch              Output
               MOV       AL,#$08             Backup
               CALL      ?putch              Output
               DEC       SI                  Backup pointer
               DEC       CX                  Reduce length
               JMP       <?2                 And proceed
* Newline, terminate entry
?4             CALL      ?putch              Echo it
               MOV       >[SI],#0            Zero terminate
               MOV       AX,CX               Return length
               RET
$DD:?COMport 2
```

Why do some labels start with question marks? They mark local branch targets that will not be called from code outside this file. Refer to the Micro-C documentation for more details on the special treatment applied to those labels by the assembler.

Say "Hello!"

Back before C compilers (excuse me, Application Development Environments) required Windows 95, came on CD-ROMs, and soaked up hard disk space denominated in tenths of gigabytes, the first C program had one essential line:

```
printf("Hello, world!\n");
```

Fortunately for us, Micro-C still retains that charming simplicity. Hello.C, shown in Listing 5, uses `printf()` to send out the obligatory message through the target system's serial I/O routines described above.

Our Hello.C program must not "fall out the bottom" of its main loop, because there is no operating system to regain control. As you'll see throughout this book, essentially all our embedded programs run continuously from power-on to shutdown. High octane embedded systems require real operating systems with dynamically loaded programs, but we don't need that level of performance just to explore the ISA bus.

Hello.C uses a `while` loop to hold the do-forever code. In this case we increment a counter, display the value, and call a BIOS function to waste about half a second.

Listing 5
The canonical First C Program prints "Hello, world!" and returns to the operating system. Our ISA-bus target system doesn't have an operating system, so, after this version of Hello.C displays its message, it begins an endless loop showing the contents of a simple counter. A BIOS function provides a half-second delay between displays.

```
#include <8086io.h>
#include "e:\mc86\custom\8086base.h"

#define LOOPDELAY 8                    /* units of 64K microseconds         */

unsigned int Counter;

main() {

   serinit(9600,1);                    /* set up serial port                */
   putstr("Hello, world!\n");
   Counter = 0;

   while (TRUE) {
      printf("Counter: %5u\r",++Counter);
   asm {
      MOV   CX,#LOOPDELAY
      MOV   DX,#0
      MOV   AH,#$86
      INT   $15
   }
   }
```

The delay is essential because there is no point in sending continuous 9600 b/s data: each line would take only about 10 milliseconds, and you just can't read text off the screen that fast.

Micro-C includes a *Command Coordinator* that runs the preprocessors, compiler, assembler, linker, and so forth in the right order with the appropriate files, as well as a character-based Integrated Development Environment. I prefer the command-line compiler, mostly because that's what I started out with, and use this command line (in a batch file, of course) to recompile the program on the host system:

```
cc86 hello -ciklmop m=t
```

The result appears in file Hello.HEX with Intel HEX format. Boot MON86 from the floppy in the target system, then use the **Load** command. Send Hello.HEX from your host system to the target system using your comm program's ASCII file transfer function, then do a **G 0100** to start Hello.

After you're comfortable with Micro-C, you may want to omit some of the compiler switches. The **-k** option keeps all the intermediate files, generating quite a clutter in your \temp subdirectory, and is of use mostly while you familiarize yourself with the compiler. The **-c** option includes all of your C language comments in the assembler output, which can inhibit the peephole optimizer's best efforts.

Because MON86 uses COM2 for debugging output and Hello uses COM1 for normal output, you must have two serial ports on your host PC to monitor the action. This is one of the big advantages of a multitasking operating system like Windows 95 or OS/2: simply run two cables, start two comm program sessions, and watch their windows update simultaneously.

You can also use a PC with a single port and one comm program, but you must switch the cable from the target system's COM2 port to COM1 after starting Hello. You'll miss the `Hello, world!` output message, but the counter values should come through correctly and display on your host system.

You can use MON86 to step through your C programs at the assembler level, set breakpoints, and dump memory. After you're sure the code is solid, use HEXFMT to convert the HEX file to a COM file just as you did for Rotate.HEX:

```
hexfmt hello.hex -b w=hello.bin
```

Copy that file to a diskette with the boot loader from Chapter 1, pop it in the target system's drive, and hit the Reset button. You should see the same output from COM1 as you did when running it under MON86.

At this point you have everything required to develop moderately complex embedded programs running on a standard *x*86 system board. The diskette boot loader reads a program from disk into the target system's RAM without using an operating system or specialized development hardware. Micro-C provides the programming essentials in a well documented, comprehensible package. Your program sends trace information through the serial port to your host system.

OK, ball's in your court…

Release Notes

The files for this chapter include all the demo program source code, `HEX`, and `BIN` files, as well as the serial port code in `SerIOFF.ASM`. There are several useful batch file to set up Micro-C's environment variables, too. See the `ReadMe.txt` files in both this chapter's subdirectory and the `\ISABus\Code\Micro-C` subdirectory.

You'll need Dunfield's 8086 Micro-C Developer's Kit to compile and assemble the source code. The debuggers described above are a part of the package, as well as numerous utility programs and debugging features that I haven't mentioned. See the Sources appendix for pointers to the DDS Web page.

Of course, you might prefer a different compiler. The disadvantage of the Borland, Microsoft, and similar high-end PC C compilers is that they are primarily intended for use with DOS and Windows. Adapting them to a DOS-less embedded system is possible, but it's not a trivial exercise. We'll explore this topic in much more detail in Chapter 11.

However, if you already have a favorite *x*86 C compiler, take a look at its `Tiny` model startup code and see what you can do. You must verify that the generated code has no DOS or Windows calls, which probably rules out the normal character I/O routines (if not all the startup code), and write your program without using any DOS-based library routines. Give it a try and see how it works!