11 Beyond Small

Does your application suffer from Creeping Featuritis? The symptoms include a `ReadMe` file bigger than the program, multiple layers of nested functions, and a tendency to display odd behavior should you dare use any new features. You'll know it when you see it…

One of the (few) nice things about 8031 microcontroller projects is how the CPU's limited address space puts an upper limit on complexity. Apart from obvious perversions, like paging 2 MB of code into an 8031 (yes, it's been done), you can only do so much with a single-chip microcontroller. This chapter may herald the end of such innocence, as embedded PCs, those computers in controllers' clothing, have no such limitations.

Thus far, we've used Dunfield Development Systems' Micro-C, a compiler admirably suited to embedded projects. The startup code occupies only a few lines, it has simple and well controlled memory requirements, and, best of all, it works! You also get a diskette of video, serial, joystick, and other PC-flavored functions that help jumpstart your embedded PC projects.

But a recurring question runs, "So, what about our Borland and Microsoft C compilers?" Suddenly, life becomes more complicated.

Unlike Micro-C, which was designed for small controller projects, the Borland and Microsoft C compilers grew up with the PC. By necessity, they include the assortment of memory models required for the 80*x*86's segmented, real-mode architecture. Whether those complex convolutions are a Good Thing or not remains open to debate, but, if you must crawl into the PC's hidden spaces, these compilers help you get there.

Those compilers have a catch: perforce, mainline C/C++ compilers for the PC produce code that must run under DOS, Windows, or worse. Loading their `EXE` output files isn't easy, their C startup code goes on for pages, and operating system functions litter their startup and runtime library files. All that makes producing a standalone program for a DOS-less PC a task for the stoutest of heart.

Although several authors have tackled the subject of directly embedding DOS programs, I believe that remains an unrewarding and endless task. Each new version of the compiler ripples changes through your existing applications, making you spend time and effort keeping up with the Borlands and Microsofts of the world. Basically, all you *really* want is to sprinkle magic dust on your `EXE` file and turn it into an embedded PC file.

In this chapter, we'll cover some of the issues involved in producing DOS-less projects using a DOS compiler, just so you know what you're missing with Micro-C. I'll also introduce Locate, a commercial product from Paradigm Systems, that helps solves this problem. There are similar products available from other companies, so check the Sources appendix, peruse the ads in your favorite magazines, and troll the Web for the current state of the art.

Paradigm allowed me to include a customized version of their TDREM program on the diskette accompanying this book. If you've built a Firmware Development Board with battery backed RAM or (E)EPROM, you can debug your Borland C code using Turbo Debugger. The ReadMe.txt file for this chapter includes the ugly details and relevant versions.

Along the Mainline

After years of working on large-system software, a friend recently began writing PC code. As he put it, "C is C, but what the heck is all this about NEAR and FAR pointers? When do I need LARGE memory model? What's going on here anyway? Who's responsible for this outrage?"

As long as you don't need more than 64 KB of memory, PC programming can be easy, because the familiar COM file format with its single segment does nearly everything you need. As you've seen, a COM file holds a binary image of the program's code and initialized data. Running the program requires nothing more than copying the disk file into RAM, setting the segment registers, and passing control to the first instruction at offset 0100 in the code segment.

But, when you must create complex programs or manipulate megabytes of data in real mode, the Intel *x*86 Segmented Memory Monster rears its ugly head. Because the simple COM file, left over from days when 64 KB was a *lot* of RAM, has no way to specify multiple program or data segments, Microsoft's DOS designers came up with the EXE file. It's not just a different file format, but a completely different way to treat executable program files.

Here's the gotcha: programs with multiple code or data segments *must* come in EXE-format files. Even something as simple as a SMALL model program with code in one segment and data in another requires the full EXE file structure. Worse yet, because of the additional segment information, EXE files for anything other than TINY model programs cannot be converted directly into COM files.

Figure 1 shows the fields in a DOS EXE file header. The Code and Data Segments field near the bottom contains the actual instructions and variables created from your source code, similar to the stuff that's normally in the COM file. The remaining

fields define the overhead information required to load those segments into RAM, adjust them for the actual memory addresses, set up the CPU registers, and start executing the program.

The compiler and linker produce the `EXE` file, but they cannot assign the final segment addresses. That step occurs when DOS allocates memory and sets the load address just before reading the file from disk. The Relocation Table entries point to

Figure 1
Programs stored as EXE files have a header with additional information used to load and run the program. The exact format depends on the DOS version, but older files will generally work unchanged on newer systems. In this context, a paragraph is 16 bytes of storage that starts on a 16-byte boundary.

Offset	Description
0000	Signature: MZ
0002	File length modulo 512 (remainder)
0004	File size in 512-byte units (rounded up)
0006	Number of relocation table entries
0008	Header size in paragraphs
000A	Minimum RAM needed in paragraphs
000C	Maximum RAM needed in paragraphs
000E	Location of stack segment in paragraph
0010	Initial SP value
0012	Checksum (not used!)
0014	Initial IP value
0016	Location of code in paragraphs
0018	Offset of relocation table in bytes
001A	Overlay number
001C	Optional reserved space
...	Relocation table
...	Optional reserved space
...	Code and data segments
...	Stack segment

the values within the code and data that require some adjustment before execution. In effect, the DOS EXE file loader performs the final link step that puts a relocatable program at an absolute address.

For example, the C statement

```
++Variable;
```

might compile into the assembly language instruction:

```
INC Variable
```

The DS register, implicit in that INC instruction's operation, must hold Variable's segment address. But the DS value depends on precisely where Variable wound up when DOS loaded the file. If your program accesses more than 64 KB of data, each variable has a corresponding DS and the compiler must insert code that loads the proper value into DS before accessing each one. All that fussing with segment registers helps make LARGE model programs run slower than you might expect.

Unlike DOS programs, many of our embedded programs are definitely *not* relocatable. For example, if our BIOS extensions in Chapter 10 didn't reside at precisely C800:0000, they simply wouldn't work. The EXE file format's flexibility becomes a liability in our situation.

Therefore, we must find a way to assign absolute segment addresses and convert the EXE file into an EPROM image or a binary disk file. That's part of what Locate does for a living, but there's more to the story.

It's worth noting that some protected mode operating systems can load and run code without making any relocation changes. The trick lies in their use of what's called FLAT memory model, which takes advantage of the memory management hardware in '386 and higher CPUs. In effect, a FLAT model program uses 32-bit addresses in a single segment that can span up to 4 GB. We can't use this in real mode, but imagine what a COM program would look like without its 64 KB addressing limitation.

Jumping the Mainline Track

By now, you should be familiar with the C startup code that gets control before the first line of your main() function. Micro-C's startup code requires only a few lines of assembler and we have modified it to do some interesting things. With a little tweaking, we even turned a more-or-less ordinary Micro-C program into a workable BIOS extension.

The Borland or Microsoft C startup code is considerably more elaborate than anything we've seen so far. In fact, the Borland C++ 3.1 startup file spans 746 lines and handles everything from clearing variables and initializing the file system to parsing the DOS environment strings. Quite a bit of that effort does us no good at all (no DOS means no DOS environment strings) and some of it can be actively harmful in an embedded environment.

And then there's Borland C++ 5.0, even bigger and more complex… which is probably obsolete by the time you read this. Get the point?

The startup code need do nothing more than the program requires. Because we won't use the DOS file system, all that code can simply Go Away. Similarly, with no DOS environment or command line information, eliminating that part of the startup routine makes perfect sense.

You cannot discard other sections quite so blithely. The startup code calls many DOS functions to adjust the program's stack and heap, return excess storage to the operating system, save and install interrupt vectors, and even display error messages. You must replace each of these functions with code that performs the same operations without invoking DOS: running a C program without a heap, for example, just isn't possible. Even if *you* don't call DOS services to allocate a heap, *someone* must do the job before your program reaches its first `malloc()`.

On top of this, the startup code handles such minutiae as defining the proper segment order for the linker, calling C/C++ entry and exit functions, and so forth. Some of these routines tie tightly into the way the compiler generates code, allowing well intentioned (but misinformed) changes to wreak subtle havoc on your program. The requirements of C++, which I won't attempt to cover, add yet another layer of complexity beyond what you expect for a C program.

The good news: `Locate` includes a sample C/C++ startup code file that you can use unchanged. Comparing the startup files from Borland and Paradigm tells me I'm glad I didn't have to figure out those changes on my own. Try it yourself and see.

The last vestige of DOS dependence lurks in the runtime library. Many C library functions use DOS calls, but, unless you have the library source code and plenty of time, you cannot tell which routines pose a problem. Because the library invokes DOS functions through software interrupts, the linker cannot identify nonexistent functions. Your code simply crashes when an `INT 21h` branches into the weeds.

The `Locate` package includes a program that creates a new set of libraries by deleting all the DOS-dependent routines from the standard Borland C runtime library files. If the linker complains about "missing functions" and "unresolved

externals" in your program when you use those libraries, well, even if you must write some additional code, that should be considerably better than debugging a mysterious failure and *then* writing the missing code.

Now we have all the pieces in place: absolute address assignment, a tailored startup file, and a purified runtime library. The rest of the task should be a matter of turning the crank…

Embedding a Sieve

Listing 1 shows Paradigm's version of the venerable `Sieve` benchmark program. I've added a line that displays the loop counter on the Firmware Development Board's LEDs and increased the test limit to 30,000 integers. I did not actually verify that the program produces the right answers… in this situation, it really doesn't matter.

Listing 1

The Paradigm Locate disk includes the good old Sieve benchmark program. I've modified it to display the iteration number in raw binary on the Firmware Development Board's LED display so you can watch it run. The key point: this code doesn't look any different than any other C program, because the startup code and Locate handle all the hocus pocus under the covers.

```
#define TRUE 1
#define FALSE 0
#define SIZE 30000

#include <dos.h>
char flags[ SIZE + 1 ];

void main(void)
{
   unsigned int i, k, count, loops;
   loops = 0;

/* Run the Sieve forever */
   for ( ; ; ) {

/* Perform the initialization */
      count = 0 ;
      for (i = 0; i <= SIZE; i++)
         flags[i] = TRUE ;

/* Run the Sieve */
      for (i = 2; i <= SIZE; i++) {
         if (flags[il) {
/* Cancel out all multiples of this prime */
            for (k = i + i; k <= SIZE; k += i)
             flags[k] = FALSE ;
            count++ ;
         }
      }
   outport(0x031e,~loops++);          // display count on FDB LED segments
   }
}
```

The code looks just like any other C program you've ever seen, because all the magic that embeds it into a DOS-less PC occurs elsewhere. The `#include <dos.h>` line might seem worrisome, but that file simply defines the `outport` macro. Fear not, your program won't invoke DOS just for that.

Compiling and linking the file proceeds normally, with one exception: you must link the object file with Paradigm's special startup code and runtime libraries. The linker produces, as usual, an ordinary Sieve.EXE file, which the MakeFile renames to Sieve.ROM. That step prevents you from accidentally running the program, should you type Sieve at the DOS command prompt.

In our case, the `outport` macro becomes an inline routine and Sieve doesn't use any library functions at all. Incidentally, because the Locate installation program creates the DOS-less libraries without changing the original Borland libraries, you can continue to build ordinary DOS and Windows programs as usual.

Locate translates Sieve.ROM into Sieve.BIN by assigning absolute memory addresses to all the segments. Locate can produce several different output formats, each with an array of suboptions. Among the choices: Intel HEX, extended HEX, Tektronix HEX, a special Absolute EXE, Intel's OMF86 files, and raw binary.

For some embedded systems projects, you may burn a HEX file into an EPROM or load a binary file into an emulator. For our work, we can use a slightly modified version of the diskette boot loader for the new files. The only change involves setting the program's initial `CS:IP` value so that our loader properly passes control to the program.

The loader we've been using so far follows the DOS COM file convention by passing control to the program at offset 0100 in the segment indicated by `CS`. All COM files expect that starting value, because they have no way to specify any other register contents. Although EXE file headers include the program's initial `IP` value (see Figure 1), the pure binary file produced by Locate no longer has that header information and, thus, gives us no clue as to where to start it up.

The C startup code expects to receive control at label `_startup`, which could be anywhere in the code segment. Paradigm's startup code positions that label and its corresponding instruction as the first byte of the file. By linking that file first, it appears at the beginning of the code segment. Then, we just tell Locate to put the code segment at the start of the output file. Because `_startup` marks the first instruction in the startup code, it becomes the first byte in the final binary image.

However, with `_startup` at that position, `IP` must be 0000 rather than 0100. Our familiar COM file loader from Chapter 1 can't handle Locate's converted EXE files!

Fortunately, we control the disk boot code. The files for this chapter include a slightly modified `BootSect` program that loads the file at address 10000, rather than 11000, then jumps to the first byte with `CS:IP` set to 1000:0000. Even though the Paradigm startup code handles all the other segment register initializations, I left the `COM`-style setup in place to give the registers a known value on entry to the startup code.

You might think twiddling the startup code to make the offset of the first instruction equal 0100, perhaps by moving other instructions ahead of it, would let you use the old loader. Remember that the `COM` loader preset `CS:IP` to address the first byte of the file at `CS:0100`, not `CS:0000`. The loader will transfer control to the first byte, even if it's not the program's entry point. Both the startup code and its loader must agree on where the first instruction resides.

You could, I suppose, add a few lines to the startup routine that catches the boot loader entry at the first byte, then reloads `CS` and `IP`. Rather than do that, I figure

Listing 2
The information in this configuration file tells Locate how to convert Sieve.ROM from EXE to binary. The MAP statements define the system address space, while the CLASS statements set the segment starting addresses. The DUP statement creates a copy of the initialized variables in the code segment, so the startup routine can copy them to the actual data segment in RAM.

```
hexfile binary offset=10000h size=8       // boot loader bin-file

listfile segments                         // output listing file

map 0x00000 to 0x005ff as reserved        // Int vectors BIOS data
map 0x0C600 to 0x0ffff as reserved        // Unused
map 0x10000 to 0x1ffff as rdonly          // 64 K application boot load area
map 0x20000 to 0x9ffff as rdwr            // Available for app
map 0xA0000 to 0xfffff as reserved        // Video, BIOS ROMS, etc

dup           DATA ROMDATA                // Copy initialized data

class         CODE = 0x1000               // Boot loader area
class         DATA = 0x2000               // Data area

order         DATA                    \   // RAM organization
              BSS BSSEND              \
              STACK

order         CODE                    \   // ROM organization
              FAR_DATA ENDFAR_DATA    \
              INITDATA EXITDATA       \
              ROMDATA ENDROMDATA

output        CODE                    \   // Output classes
              FAR_DATA ENDFAR_DATA    \
              INITDATA EXITDATA       \
              ROMDATA ENDROMDATA
```

you'll use either Micro-C's COM files or Paradigm's converted EXE files, but not both at once. Just use the loader that matches your situation and be done with it. Make *absolutely* sure you mark the diskettes while you're experimenting, though, because half of the possible combinations simply won't work.

With that as background, Listing 2 shows the segment address assignments defined in the Sieve.CFG file. Locate has far more options than I have room to describe here, but you should get a feel for the program's flexibility.

The HEXFILE statement specifies an 8 KB binary output file containing the instructions and data starting at absolute address 10000, the load point set by our diskette boot loader. You may use additional HEXFILE statements to create multiple output files during the same pass. Locate will warn you if the file includes no information at all, which can catch stupid addressing mistakes.

The MAP statements define how the program will use the CPU's address space. Locate verifies that all of the segments seem appropriate for their addresses. For example, you should not put a code segment into a read/write area that allows code overwrites. Remember that Locate and MAP cannot actually write protect your system's RAM. They simply document your intentions and verify that no other segments encroach on the code during the file conversion process.

The MAP statement provides more benefit with the TDREM debugger I'll discuss shortly. Because we load our code from disk into RAM, the rdonly (read-only) sections will be just as volatile as any other RAM. However, the debugger can detect changes within those areas and report problems.

The DUP statement creates a copy of the data segment in the ROMDATA part of the code segment. Paradigm's C startup code automatically copies ROMDATA from the code segment, where the loader put it from diskette, into the data segment. That's all it takes to get initialized variables working.

The two CLASS statements specify the starting addresses for code and data segments. The name CLASS comes about because the compiler and linker can combine several different program segments into a single physical memory segment, thus treating the original segments as a class. This statement sets the address of the first segment in the resulting class. If your application has multiple classes, you must include a CLASS statement for each one.

The ORDER statements specify how to combine individual segments into an overall class that a CLASS statement will nail at a specific address. This corresponds (roughly) with the way the linker handles groups of segments, but requires manual intervention to get it right. The Locate documentation goes into more detail on this

issue. Suffice it to say that Paradigm's sample files give you a good idea of what to do in several different situations.

Despite the seeming complexity, once you get a configuration file set up, you probably won't tweak it again until you change your program's segment structure or the target system's memory map. In fact, you can probably use the same `CFG` file for most of your embedded PC projects with little change.

Actually running `Locate` requires nothing more than another `MakeFile` line, which should take even less thought than figuring out your compiler or linker switches. While I was inside the `MakeFile`, I added a few commands that copy the resulting `BIN` file to the boot diskette. Two keystrokes in my host's DOS window now rebuild the whole application and set up the diskette for the target machine.

Although I compiled `Sieve.c` with the `Small` memory model, `Locate` and the modified runtime libraries support all models except `Tiny`. Because our custom diskette boot loader can only handle binary files up to 64 KB, you must modify the loader for truly large programs. You can certainly use my source code to get started.

Borland's `Tiny` memory model can generate `COM` files without running `Locate`, just as we did with Micro-C. Remember to eliminate all DOS functions from the startup code, link your files with a DOS-less library, and boot the result with our original (`IP` = 0100) `COM` file diskette loader.

`Locate` supports Microsoft C++ as well as the Borland compilers. I don't have any examples because, back when I developed these programs, their C compiler didn't run under OS/2 without lots of twiddling that I wasn't interested in doing. Now that I'm using Windows 95, I suppose I could switch, but it does seem like a lot of effort to get essentially the same result.

That leaves us with one question: "What if the program doesn't work?" Finding program errors is always the most difficult part of development, particularly with burn-and-crash debugging, but we now have a full-featured debugger at our disposal.

Debugging by Wire

Creeping Featuritis afflicts all PC programs. Simple command-line compilers accrete features, sprout program development utilities, and gather debuggers on their disks. More recently, these conglomerations gelled into Integrated Development Environments delivered on a CD-ROM or two. If you look hard, you can actually find the command-line compiler hidden in a subdirectory and, perhaps, described in a footnote.

Creeping Featuritis means more than never buying another diskette or figuring out what to do with a stack of obsolete CD-ROMs. Somewhere along the way, we lost sight of the fact that an IDE spawning a debugger to load a nontrivial program compiled with every debugging option turned ON can soak up all the RAM in a system. In some cases, the debugging infrastructure doesn't leave enough room for the program we intended to debug in the first place. Thus began the arms race to exploit expanded memory, extended memory, any memory except the precious 640 KB down there in DOS territory.

Borland's Turbo Debugger designers, in either a stroke of genius or an admission of defeat, included a *remote debugging* mode that runs the debugger's user interface on an entirely separate machine from the target program. A serial cable or network connection eliminates the requirement for exotic RAM mapping, video display sharing, and mouse handoffs.

Whenever Turbo Debugger refers to a memory location, an I/O port, or a CPU register, it sends a message from the host system across the cable to the target system. A debugging kernel in the target handles the request and returns the results over the same cable to the host. Because the kernel doesn't interact with the target's keyboard, video display, or mouse, it can be both small and simple. Most of the conflicts between the target program and the debugger simply Go Away when the two programs run on different systems.

Windows has, to some extent, relieved the DOS memory shortage on '386 and higher CPUs by putting the PC's entire multimegabyte RAM address space to good use. Unfortunately, because high capacity debuggers exact a significant toll in system overhead and, of course, our target program don't run under Windows, you will still find a remote debugger your best choice.

The folks at Paradigm (and others, as well) figured out that remote debugging could be the solution for debugging embedded *x*86 systems. With a bit of programming magic, they can fool Turbo Debugger into thinking it has an entire PC on the other end of the wire, even if the target system lacks disks, DOS, or even a BIOS to support the embedded program.

Adapting `TDREM` to the Firmware Development Board was straightforward, as the Paradigm code can handle most hardware configurations with just a few `EQU` or `#define` twiddles. I picked the simplest target interface: polled communication through the COM1 serial port.

Because `TDREM` should get control before booting the target program from disk, I turned it into a BIOS extension using code similar to that shown in Chapters 8 and 10. The extension code hooks `INT 19h` and returns to the BIOS. When the BIOS

finishes initializing everything, it invokes `INT 18h`, which starts the `TDREM` remote interface. My code shows its progress on the Firmware Development Board LEDs, concluding with a cheery **`td`** immediately before diving into Paradigm's Turbo Debugger support routines.

Installing the extension uses the same BIOS extension loader program we used earlier. It copies `TDREM` from diskette into the FDB's battery backed RAM and computes the checksum required for a valid BIOS extension. Remember to hold down the FDB's pushbutton while booting the loader to install the extension.

After that, running the debugger requires just booting the target system. My prologue code checks the FDB's pushbutton before starting the extension. With the button pressed, control returns to the BIOS without capturing `INT 19h` and the standard diskette boot sequence occurs. The LEDs show a pair of dashes to indicate that the extension did not install itself.

With the pushbutton up, the extension installs itself, the FDB LEDs show **`td`**, and you can start Turbo Debugger on your host system. For the DOS version of TD, use the command line:

```
TD -RS3
```

on the host to specify a 38.4 kb/s data rate. I found that `TDREM`'s default 115 kb/s rate doesn't work on my target system, probably because the serial bytes arrive too fast for the software polling loop. Obviously, an interrupt-driven interface would solve that problem, but fitting it into the target system involved more complexity than I thought justified for this part of the project.

The `TDREM` kernel always tells the debugger that it has an outdated copy of the program because, in fact, it doesn't *have* a copy. Turbo Debugger then asks whether you want to send the "newer" version… you should always answer Yes.

Now you can choose either Micro-C or a mainline C compiler, depending on your project's goals and finances. Either compiler will work for a broad range of projects, so there shouldn't be much holding you back. Go for it!

The Second-Worst Hack

While we're on the topic of debuggers, here's an interesting application of a deliberate error. Back in Chapter 8, I described the Worst Hack in PC-dom: using the keyboard controller to reset an 80286 CPU and bail out of protected mode. While doing some research for another project, I came across a tidbit that probably qualifies as the Second-Worst Hack in PC-dom.

Suppose you're writing a protected-mode operating system (can you spell OS/2?) for '286 systems and realize that the Worst Hack's speed (or lack thereof) will definitely clobber overall system performance. What to do?

Easy: crash the system!

Starting with the 80286, Intel CPUs detect severe errors and shut down, rather than continuing with unpredictable results. For example, when `SP` = 0001 a `PUSH` instruction shuts down the CPU, instead of wrapping `SP` to FFFF and clobbering RAM outside the stack area (as the 8086 and 8088 did). The CPU sends a specific status signal that tells the rest of the system what happened, then waits for a hardware reset or an **`NMI`**. It does not execute any further instructions, process external interrupts, or do anything else besides wait quietly.

Rather than leave the system stalled forever, the Original IBM PC AT system board (and, thus, all subsequent PCs) included a shutdown detection circuit that causes a hardware reset. The CPU pops out of reset in real mode, the BIOS checks the Real-Time Clock's battery backed RAM for the shutdown reason code, and vectors to the appropriate routine. Apart from the fact that the reset comes from a different circuit, it works just as you saw in Chapter 8.

Because the reset happens at hardware rather than software (or even firmware) speeds, the whole sequence takes a tiny fraction of a millisecond. That's enough faster than the Worst Hack to make it worthwhile. It's even faster than those hyperthyroid keyboard controllers with hardwired command bypasses, too.

How do you force a CPU shutdown? The CPU invokes a Double Fault interrupt handler that gets control when two protection violations occur on a single instruction. If the Double Fault handler *also* causes a violation, the CPU shuts down. It seems the OS/2 designers set up a deliberate *triple* fault to bail out of '286 protected mode faster than the standard keyboard controller command.

Although I don't know the exact method they used, you could mark a segment as Not Present, then invalidate both the Segment Not Present and Double Fault interrupt gates. You disable external interrupts and the **`NMI`** input, aim a segment register at the missing segment, and fetch a byte. The fetch triggers a Segment Not Present interrupt through `INT 0Bh`, where the invalid gate causes a Double Fault interrupt (`INT 08h`), and the invalid Double Fault gate slam dunks a triple fault.

Thud!

Release Notes

The Sieve binary files require the modified BootSect diskette loaders from this chapter. You'll find the source and binary files for all four diskette sizes in the subdirectory: as always, make sure you pick the appropriate one for your system.

Because this chapter's version of Paradigm's TDREM remote driver for Turbo Debugger runs as a BIOS extension, copy it to a diskette along with the appropriate LoadExt extension loader from Chapter 8. Boot that diskette in your target system while holding the pushbutton down to stuff TDREM into the Firmware Development Board's battery backed RAM. The next boot, with the button up, will display **td** on the LEDs when the target is ready for Turbo Debugger.

TDREM will use polled operation through COM1 at 38.4 kb/s. Remember to start Turbo Debugger on your host PC using the -RS3 switch to get the proper serial data rate. All the usual features should be available, except breaking into a running program. That requires a fully interrupt-driven configuration that I'll leave as an exercise for you. Don't forget that watchdog timer…

I assume you already have the Borland or Microsoft compilers running on your system. Follow the directions accompanying Pardigm's programs to properly install Locate and modify your runtime libraries. Remember that few, if any, embedded PC programs are upwardly compatible with new compilers; consult the ReadMe.txt file for the versions you'll find on the diskette with this book.

Note that the software for this book does *not* include Paradigm's Locate program or their other utilities and libraries. You may use the executable programs found on the diskette, but you must have the complete Paradigm Locate package to modify and recompile them.

If your finances won't stretch around a commercial Locate program, you should spend some time searching the Internet for shareware equivalents. Several magazine Web sites have large collections of links: *Circuit Cellar INK* at **www.circellar.com**, *C/C++ User's Journal* at **www.cuj.com**, and *Embedded Systems Programming* at **www.embedded.com**. Dunfield Development Systems at **www.dunfield.com** will conduct you to several useful programs for low-level embedded work. You'll find other pointers in the Sources appendix.