John Socha and Peter Norton

ASSEMBLY LANGUAGE FOR THE PC

THIRD EDITION

- Harness the power of the 80x86 microprocessor
- Build real-world programs, including a general-purpose, full-screen disk editor
- Add compact assembly code to your C and C++ programs
- Use assembly code in protected-mode and Windows programming
- Master advanced techniques such as step-wise refinement and modular design

Disk Inside!

Disk includes all the code examples in the book plus more than 30 valuable, time-saving assembly routines for your C and C++ programs

Assembly Language for the PC, Third Edition

John Socha
and
Peter Norton

Brady Publishing

New York London Toronto Sydney Tokyo Singapore

Brady Publishing
A Division of Prentice Hall Computer Publishing
15 Columbus Circle
New York, NY 10023

ISBN: 1-56686-016-4

Library of Congress Catalog No.: 92-44830

Printing Code: The rightmost double-digit number is the year of the book's printing; the rightmost single-digit number is the number of the book's printing. For example, 92-1 shows that the first printing of the book occurred in 1992.

95 94 93 4 3

Manufactured in the United States of America

About the Authors

John Socha is known in the PC industry for his writing and his software products. In the early days of the IBM PC, he wrote a column for the now defunct magazine *Softalk*, in which he published such programs as ScrnSave (the first screen saver) and Whereis (the first program to find files on hard disks). After the demise of *Softalk*, John concentrated on finishing his Ph.D. in Physics and writing The Norton Commander, which became a best-selling software product. He is also the coauthor of the best-selling book *PC-World DOS 5 Complete Handbook* and the author of *Learn Programming* and *Visual Basic 2 with John Socha.* John grew up in Wisconsin, earned a B.S. degree in Electrical Engineering from the University of Wisconsin, and an M.S. and a Ph.D. in Applied Physics from Cornell University. He now runs a software company called Socha Computing that develops Windows and Macintosh utility software in Kirkland, Washington.

Entrepreneurial software developer Peter Norton is the author of more than a dozen technical books. His first book, the best-selling classic *Inside the IBM PC*, was published in 1983, and was followed later in the year by another Brady book, his immensely popular guide to DOS.

Born in 1943 and raised in Seattle, Washington, Mr. Norton attended Reed College in Portland, Oregon, where he majored in physics and mathematics. He completed his education at the University of California at Berkeley where he received a Bachelor of Arts in mathematics. During the Vietnam War he served in the U.S. Army and taught emergency medicine to combat medics. He received a priestly teaching certificate after studying for five years in a Zen Buddhist monastery. Mr. Norton held a variety of positions in the computer industry before he created the Norton Utilities software program in 1982. He subsequently founded his own company, Peter Norton Computing, Inc., to develop and market his software. During the last decade, he has written many popular columns for computer magazines. In 1990 he sold his company to Symantec Corporation and devoted himself to the Norton Family Foundation, a fund that provides financial assistance to arts organizations in the Los Angeles area. He currently lives in Santa Monica, California, with his wife Eileen and their two young children Diana and Michael.

Credits

Publisher
Michael Violano

Managing Editor
Kelly D. Dobbs

Editor
Tracy Smith

Production Editors
Bettina Versaci
Tom Dillon

Developmental Editor
Perry King

Editorial Assistant
Lisa Rose

Book Designer
Michele Laseau

Cover Designer
HUB Graphics

Production Team
Katy Bodenmiller
Christine Cook
Lisa Daugherty
Carla Hall-Batton
Dennis Clay Hager
Howard Jones
John Kane
Loren Christopher Malloy
Sean Medlock
Tim Montgomery
Roger Morgan
Linda Quigley
Michelle Self
Susan Shepard
Angie Trzepacz

Contents

Introduction

Why Learn Assembly Language?

There are now more reasons than ever to learn assembly language. Even programmers who only use a high-level language like C/C++ or Pascal can benefit from learning assembly language.

When we wrote the first edition of this book, several years ago, programmers and companies were actually writing entire programs in assembly language. For example, Lotus 1-2-3 before release 2 was written entirely in assembly language. These days, very few programs are written entirely in assembly language. Even 1-2-3 has now been rewritten in C, and many programmers are moving on to C++, which is a very powerful descendent of the C programming language. So if people are not writing large programs in assembly language, then why would you want to learn and use assembly language?

Knowing assembly language will make you a better programmer, even if you never actually write any assembly-language subroutines or code for your programs. Many of the idiosyncrasies you will find in C/C++ or Pascal programs have their roots in the microprocessor and its design. When you learn assembly language, you are really learning about the architecture of your computer. This insight should make many of the artifacts of computer languages clear to you. For example, you will see why an integer has a value range of –32,768 to 32,767, and you will learn all about bytes, words, and segments.

Assembly language programs are at the heart of any PC compatible computer. In relation to all other programming languages, assembly language is the lowest common denominator. It takes you closer to the machine than the higher-level languages do, so learning assembly language also means learning to understand the 80x86 microprocessor inside your computer.

This knowledge about the microprocessor is also very useful in the world of Microsoft Windows, as you will see in Chapter 33, which discusses protected-mode and Windows programming. You will learn about some features of Windows programming that you probably won't find in any other book. This is because there are programming techniques that you can do and learn about in assembly language that you would not normally learn about in a high-level language.

At some time or other most programmers write or modify some assembly language code. This is even true for Microsoft Windows programs. The most common reason to use assembly language is for speed. Computers, modern C/C++ and Pascal compilers are very fast; compiled code runs very quickly. But there are times when even compiled C/C++ or Pascal code won't be fast enough. Very often in these cases you can write a small amount of assembly-language code to perform the work in the inner loop.

Compilers have also improved significantly since we wrote the first edition of this book. Years ago you *had* to use an assembler, such as Microsoft's Macro Assembler, to add assembly-language code to your programs. But almost all compilers these days support *in-line assembly*, which allows you to add a few lines of assembly-language code directly to your program, between lines of C/C++ or Pascal code. This makes it much easier to use small amounts of assembly language in your programs. You will learn how to use in-line assembly in Chapter 31.

There are also other cases which require programmers to write larger amounts of code in assembly language. If you are writing a device driver for DOS or Windows, or almost any other operating system (except for Windows NT), you will probably have to write the device driver in assembly language. For example, Windows has a very powerful mechanism called a Virtual Device Driver (also known as a VxD) that allows you to alter the way devices work inside Windows for all Windows and DOS programs running under Windows. These VxDs must be written in assembly language.

The Approach We Use

Like most introductory books on assembly language programming, this book shows you how to use the instructions of the 80x86 microprocessor. But we will go much farther and cover *advanced* material that you will find invaluable when you start to write your own programs.

By the time you finish reading this book, you will know how to write large assembly language programs and how to use assembly language in your C/C++ programs and Windows programs. Along the way, you will also learn many techniques that professional programmers use to make their work simpler.

These techniques, which include modular design and step-wise refinement, will double or triple your programming speed and help you write more readable and reliable programs.

The technique of step-wise refinement, in particular, takes a lot of the work out of writing complex programs. If you have ever had that sinking, where-do-I-start feeling, you will find that step-wise refinement gives you a simple and natural way to write programs. It is also fun!

We will also try to show you how comments can help you write better programs. Well-written comments explain *why* you are doing something, rather than *what* you are doing. In well-written code it should be obvious what the code is doing, but it certainly may not be clear why you are doing something.

This book is not all theory. We will also build a program called Dskpatch (for Disk Patch), and you will find it useful for several reasons. First, you will see step-wise refinement and modular design at work in a real program and you will have an opportunity to see why these techniques are so useful. Also, Dskpatch is a general-purpose, full-screen editor for disk sectors—one that you can continue to use both in whole and in part long after you have finished with this book.

Organization of This Book

We have chosen an approach to teaching assembly language that we think will get you up to speed as quickly as possible without overwhelming you with details. With this approach you will be able to learn a lot about the 80x86 instructions before you have to learn about using an assembler. We have also chosen to write most of the sample programs for DOS because it is a lot easier to write small programs in DOS than to write the same program for Windows. The concepts and techniques you learn will also apply to programs you write in other operating environments, so if you want to program for Windows, you might read Parts I and II, then skip to Chapters 31 and 33.

This book is divided into four parts, each with a different emphasis. Whether or not you know anything about microprocessors or assembly language, you will find sections that are of interest to you.

Part I focuses on the 80x86 microprocessor. You will learn the mysteries of bits, bytes, and machine language. Each of the seven chapters contains a wealth of real examples that use a program called Debug, which comes with DOS. Debug will allow us to look *inside* the 80x86 microprocessor nestled deep in your PC as it runs DOS. Part I assumes only that you have a rudimentary knowledge of programming languages and know how to work with your computer.

Part II, Chapters 8 through 16, moves on to assembly language and how to write programs for the assembler. The approach is not complicated, and rather than cover all the details of the assembler itself, we will concentrate on a set of assembler commands needed to write useful programs.

We will use the assembler to rewrite some of the programs from Part I, and then move on to begin creating Dskpatch. We will build this program slowly, in order for you to learn how to use step-wise refinement in building large programs. We will also cover techniques like modular design that help in writing clear programs. As mentioned, these techniques will simplify programming by removing some of the complexities normally associated with writing assembly-language programs.

In Part III, which includes Chapters 17 to 28, we will concentrate on using more advanced features found in PCs. These features include moving the cursor and clearing the screen. We will also discuss techniques for debugging larger assembly-language programs. Assembly-language programs grow quickly and can easily be two or more pages long without doing very much (Dskpatch will be longer). Even though we will use these debugging techniques on programs larger than a few pages, you also will find them useful with small programs.

Part IV covers a number of advanced topics that will be of interest to you when you start to write real programs. The first two chapters cover details about COM programs, memory, and segments. Then there is a chapter on writing directly to screen memory for very fast screen displays. Next, there is a chapter on writing assembly-language procedures that you can use in your C/C++ programs. You will learn how to write general-purpose routines entirely in assembly language for any memory model, and you will also learn how to use in-line assembly. There is a lot of material in this chapter that is very difficult to find elsewhere. Next is a chapter on RAM-resident programs, complete with a program called DISKLITE that adds a disk light to your screen. Finally, you will find a chapter on protected-mode and Windows programming that will show you some interesting details about the way Windows works on an 80386 or better microprocessor.

What Is on the Disk?

The disk included with this book contains most of the code examples you will find in this book in a directory called CHAPS. We have included a more advanced version of Dskpatch in a directory called ADVANCED. Appendix A discusses both of these directories.

Appendix B contains a complete listing of the Dskpatch program we will build in this book. Appendix C contains listings for some general-purpose libraries for C and C++ programs that you'll find on the disk at the back of this book.

The set of libraries for C and C++ programs allows you to write quickly to the screen and support a mouse. This code is the code that we used to write the Norton Commander, which means we had to deal with issues that other authors who have not written commercial software have never encountered. It also means that these libraries have been thoroughly tested by hundreds of thousands of users. All of this code is included on the disk at the back of this book, and you'll find complete listings of this code in Appendix C.

What You Will Need to Use the Examples

There are a few tools you will need in order to use the examples in this book. For all the examples in Part I, you need nothing more than the Debug program that comes with DOS. In Part II you will need a simple text editor, such as the Edit program in DOS 5 or later, and an assembler, such as Microsoft's Macro Assembler (MASM) or Borland's Turbo Assembler (TASM). All of the examples in this book have been tested on various versions of MASM and TASM. In Chapters 31 and 33, however, you will need MASM 6 (or later) or TASM to assemble the examples.

Finally, to compile some of the C examples in Chapter 33, you will need QuickC for Windows, Borland C++, or Microsoft C 6.0 or later. All of these C compilers support in-line assembly and allow you to compile Windows programs.

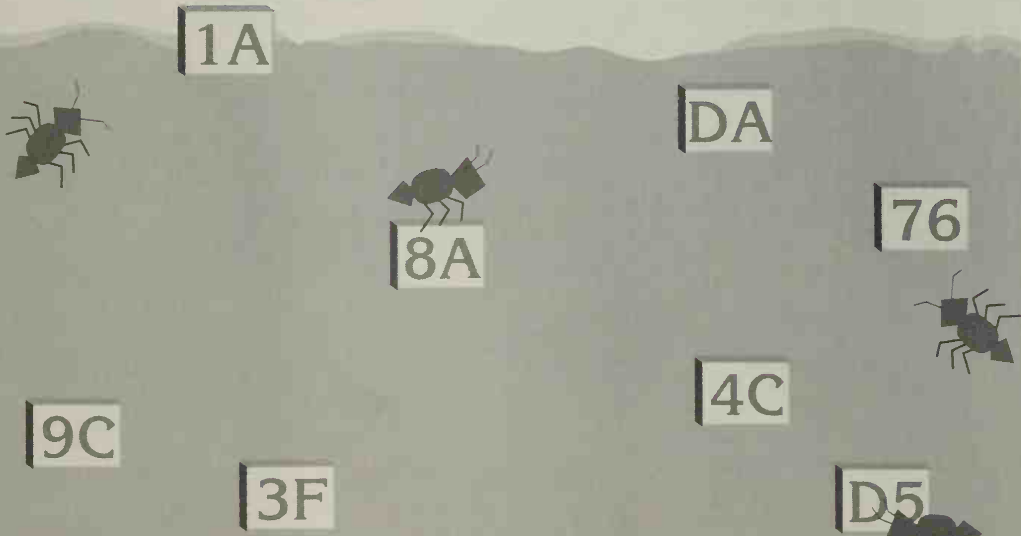
Dskpatch

In our work with assembly language, we will look directly at disk sectors and displaying characters and numbers stored there by DOS in hexadecimal notation. Dskpatch is a full-screen editor for disks and it will allow us to change these characters and numbers in a disk sector. By using Dskpatch you could, for example, look at the sector where DOS stores the directory for a disk and you could change file names or other information. Doing so is a good way to learn how DOS stores information on a disk.

You will get more out of Dskpatch than just one program, though. Dskpatch contains about 50 subroutines. Many of these are general-purpose subroutines you will find useful when you write your own programs. Thus, not only is this book an introduction to the 80x86 and assembly-language programming, it is also a source of useful subroutines.

PART I

Machine Language



CHAPTER

Learning Debug and Computer Arithmetic

In this chapter you are going to learn how the computer handles arithmetic. This is a fundamental concept for assembly-language programs. It has a significant impact on all programs you will write, whether in C, assembly language, or any other computer language.

1

Topics Covered

A Brief History of Intel Microprocessors

Counting the Computer Way

Counting with Hexadecimal Numbers

Using Debug

Doing Hex Arithmetic

Bits, Bytes, Words, and Binary Notation

Two's Complement—An Odd Sort of Negative Number

Summary

In this chapter you will start working with the microprocessor inside your computer right away. You will use a program called Debug to start working with computer arithmetic. It is a good starting point because Debug is at the heart of almost all assembly-language programs. We will continue to use Debug in all the chapters in this book, which will allow you to write and run very simple programs.

A Brief History of Intel Microprocessors

Before looking at assembly language, a few words are in order about microprocessors. Currently (as of 1992) there are four main microprocessors used in PC compatible computers: the 8088, 80286, 80386, and 80486 microprocessors (and Intel is working on the next generation, which will be called the Pentium). The 8088 microprocessor was first used in the original IBM PC and is the slowest, least powerful microprocessor. Very few computers are built with the 8088 microprocessor.

Next came the 80286 in the IBM AT, which was about four times faster, and the first computer capable of running IBM's OS/2. The 80286 is also the minimum microprocessor required to run Windows 3.1. Most computers are currently built around the faster 80386 or 80486 microprocessors, which also have additional capabilities that Windows can use.

The 80286, 80386, and 80486 are *supersets* of the 8088 microprocessor, which means any programs written for the 8088 microprocessor will run on any of the others. Almost all programs written for MS-DOS or PC-DOS, as opposed to Windows, are written using just 8088 features so they will run on all MS-DOS computers. In this book we will be covering mostly the 8088 instructions so the programs you will write will work on all MS-DOS computers. However, later in the book you will find some coverage of Windows programming in the *protected mode* of the 80286 and better microprocessors, which is used by operating systems such as Windows and OS/2. Because there are so many different microprocessors used in PCs, we will refer to the entire family as 80x86 microprocessors.

Counting the Computer Way

For your first foray into assembly language you will learn how computers count. That may sound simple enough. After all, you count to 11 by starting at one and counting up:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

A computer does not count that way. Instead, it counts to five like this: 1, 10, 11, 100, 101. The numbers 10, 11, 100, and so on are binary numbers, based on a numbering system with only two digits, 1 and 0, instead of the ten associated with our more familiar decimal numbers. Thus, the binary number 10 is equivalent to the decimal number we know as two.

We are interested in binary numbers because they are the form in which numbers are used by the 80x86 microprocessor inside your PC. But while computers thrive on binary numbers, those strings of ones and zeros can be long and cumbersome to write out. The solution? Hexadecimal numbers—a far more compact way to write binary numbers. In this chapter, you will learn both ways to write numbers: hexadecimal and binary. As you learn how computers count, you will also learn how they store numbers—bits, bytes, and words. If you already know about binary and hexadecimal numbers, bits, bytes, and words, you can skip to the chapter summary.

Counting with Hexadecimal Numbers

Since hexadecimal numbers are easier to handle than binary numbers—at least in terms of length—we will begin with hexadecimal (hex for short), and use DEBUG.COM, a program included with DOS. You will use Debug here and in later chapters to enter and run machine-language programs one instruction at a time. Like BASIC, Debug provides a nice, interactive environment. But unlike BASIC, it doesn't know decimal numbers. To Debug, the number 10 is a hexadecimal number—not ten. Since Debug only speaks in hexadecimal, you will need to learn something about hex numbers. But first, let's take a short side trip and find out a little about Debug itself.

Using Debug

Why does this program carry the name Debug? *Bugs*, in the computer world, are mistakes in a program. A working program has no bugs, while a nonworking or "limping" program has at least one bug. You can find mistakes and correct them by using Debug to run a program one instruction at a time and by watching how the program works. This is known as *debugging*, hence the name Debug.

According to computer folklore, the term debugging stems from the early days of computing—in particular, a day on which the Mark I computer at Harvard failed. After a long search, technicians found the source of their troubles: a small moth caught between the contacts of a relay. The technicians removed the moth and wrote a note in the log book about "debugging" the Mark I.

Note

From here on, in interactive sessions like this one, the text you type will be in boldface against a gray background to distinguish it from your computer's responses, as follows:

```
C>DEBUG
```

Type the gray text (DEBUG in this example), press the Enter key, and you should see a response similar to the ones we show in these sessions. You won't always see exactly the same responses because your computer probably has a different amount of memory from the computer on which we wrote this book. (We will begin to encounter such differences in the next chapter.) In addition, notice that we use uppercase letters in all examples. This is only to avoid any confusion between the lowercase letter l (el) and the number 1 (one). If you prefer, you can type all examples in lowercase letters.

Now, with those few conventions noted, start Debug by typing its name after the DOS prompt (which is C> in the following example).

```
C>DEBUG
-
```

The hyphen you see in response to your command is Debug's prompt symbol, just as C> is a DOS prompt. It means Debug is waiting for a command. To leave Debug and return to DOS, just type Q (for *Quit*) at the hyphen prompt and press Enter. Try quitting now, if you like, and then return to Debug.

```
-Q

C>DEBUG
-
```

Now we can get down to learning about hex numbers.

Doing Hex Arithmetic

We will use a Debug command called H. H is short for *Hexarithmetic*, and as its name suggests, it adds and subtracts two hex numbers. Let's see how H works by starting with 2 + 3. We know that 2 + 3 = 5 for decimal numbers. Is this true for hex numbers? Make sure you are still in Debug and, at the hyphen prompt, type the following screened text:

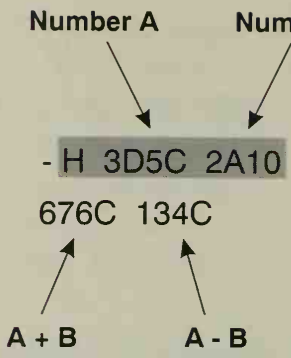
```
-H 3 2
0005  0001
```

Debug prints both the sum (0005) and the difference (0001) of 3 and 2. The Hexarithmetic command, shown in Figure 1-1, always calculates the sum and difference of two numbers, as it did here. So far, the results are the same for hex and decimal numbers: 5 is the sum of 3 + 2 in decimal, and 1 is the difference (3 – 2). Sometimes, however, you can encounter a few surprises.

For example, what if you typed H 2 3, to add and subtract two and three, instead of three and two? If you try it you get FFFF, instead of –1, for 2 – 3. Strange as it may look, however, FFFF is a number. In fact, it is hex for –1.

```
-H 2 3
0005  FFFF
```

We will come back to this rather unusual –1 shortly. But first, let's explore the realm of slightly larger numbers to see how an F can appear in a number.



```
-H 3D5C 2A10
676C 134C
```

Figure 1-1: The Hexarithmetic command.

To see what the Hexarithmetic command does with larger numbers, try nine plus one, which would give you the decimal number 10.

```
-H 9 1
000A  0008
```

Nine plus one equals A? That's right: A is the hex number for ten. Now, what if we try for an even larger number, such as 15?

```
-H 9 6
000F  0003
```

If you try other numbers between ten and fifteen, you will find 16 digits altogether—0 through F (0 through 9 and A through F), see Figure 1-2. The name hexadecimal comes from hexa- (6), plus deca- (10), which, when combined, represent 16. The digits 0 through 9 are the same in both hexadecimal and decimal; the hexadecimal digits A through F are equal to the decimals 10 through 15.

Why does Debug speak in hexadecimal? Soon you will see that you can write 256 different numbers with two hex digits. As you may already suspect, 256 also bears some relationship to the unit known as a byte, and the byte plays a major role in computers and in this book. You will find out more about bytes near the end of this chapter, but for now we will continue to concentrate on learning hex, the only number system known to Debug and hex math.

Decimal	Hex digit	
0	0	These digits are the same for both decimal and hex.
1	1	
2	2	
3	3	
4	4	
5	5	
6	6	
7	7	
8	8	
9	9	
10	A	These digits are new hexadecimal digits.
11	B	
12	C	
13	D	
14	E	
15	F	

Figure 1-2: Hexadecimal digits.

Converting Hexadecimal to Decimal

Thus far, you have learned single-digit hex numbers. Now, let's see how you represent larger hex numbers and how you convert these numbers to decimal numbers.

Just as with decimal numbers, you can build multiple-digit hex numbers by adding more digits on the left. Suppose, for example, you add the number 1 to the largest single-digit decimal number, 9. The result is a two-digit number, 10 (ten). What happens when you add 1 to the largest single-digit hex number, F? You get 10 again.

But wait—10 in hex is really 16, not 10. This could become rather confusing. We need some way to tell these two 10s apart . From now on we will place the letter h after any hex number. Thus, you can tell that 10h is hexadecimal for 16 and 10 is decimal ten.

Now let's look at how to convert numbers between hex and decimal. You know that 10h is 16, but how do you convert a larger hex number, such as D3h, converted to a decimal number without counting up to D3h from 10h? Or, how is the decimal number 173 converted to hex?

You cannot rely on Debug for help, because it cannot speak in decimal. In Chapter 10, we will write a program to convert a hex number into decimal notation so that our programs can talk to us in decimal. But right now, we will have to do these conversions by hand. We will begin by returning to the familiar world of decimal numbers.

What does the number 276 mean? In grade school, you learned that 276 means you have two hundreds, seven tens, and six ones. Or, more graphically as follows:

$$\begin{array}{lcrcr}
2 & * & 100 & = & 200 \\
\;7 & * & 10 & = & 70 \\
\;\;6 & * & 1 & = & 6 \\
\hline
276 & & & = & 276
\end{array}$$

Well, that certainly helps illustrate the meanings of those digits. Can we use the same graphic method with a hex number? Of course.

Consider the number D3h mentioned earlier. D is the hexadecimal digit 13, and there are 16 hex digits, versus 10 for decimal, so D3h is thirteen sixteens and three ones. Or, presented graphically as follows:

$$\begin{array}{lcrcrcr}
D & \rightarrow & 13 & * & 16 & = & 208 \\
\;3 & \rightarrow & 3 & * & 1 & = & 3 \\
\hline
D3h & & & & & = & 211
\end{array}$$

For the decimal number 276, you multiply digits by 100, 10, and 1; for the hex number D3, multiply digits by 16 and 1. You would multiply four decimal digits by 1000, 100, 10, and 1. Which four numbers would you use with four hex digits? For decimal, the numbers 1000, 100, 10, and 1 are all powers of 10, as follows:

$$\begin{array}{lcr}
10^3 & = & 1000 \\
10^2 & = & 100 \\
10^1 & = & 10 \\
10^0 & = & 1
\end{array}$$

You can use the same exact method for hex digits, but with powers of 16, instead of 10, so the four numbers are as follows:

$16^3 = 4096$
$16^2 = 256$
$16^1 = 16$
$16^0 = 1$

For example, you can convert 3AC8h to decimal by using these four numbers. (See Figure 1-3 for more examples.)

```
3     → 3  *  4096 = 12288
 A    → 10 *   256 =  2560
   C  → 12 *    16 =   192
    8 → 8  *     1 =     8
--------------------------
3AC8h              = 15048
```

```
7   →  7 * 16 = 112
 C  → 12 *  1 =  12
-------------------
7Ch      =      124

3    →  3 * 256 = 768
 F   → 15 *  16 = 240
  9  →  9 *   1 =   9
---------------------
3F9h      =     1,017

A      → 10 * 4,096 = 40,960
 F     → 15 *   256 =  3,840
  1    →  1 *    16 =     16
   C   → 12 *     1 =     12
----------------------------
AF1Ch         =       44,828

3      →  3 * 65,536 = 196,608
 B     → 11 *  4,096 =  45,056
  8    →  8 *    256 =   2,048
   D   → 13 *     16 =     208
    2  →  2 *      1 =       2
------------------------------
3B8D2h         =       243,922
```

Figure 1-3: More hexadecimal to decimal conversions.

Now let's discover what happens when we add hex numbers that have more than one digit. For this, we will use Debug and the numbers 3A7h and 1EDh, as follows:

```
-H 3A7 1ED
0594  01BA
```

So you see that 3A7h + 1EDh = 594h. You can check the results by converting these numbers to decimal and doing the addition (and subtraction, if you wish) in decimal form; if you are more adventurous, do the calculations directly in hex.

Five-Digit Hex Numbers

So far, hex math is quite straightforward. What happens when you try adding even larger hex numbers? Try a five-digit hex number, as follows (See Figure 1-4 for more examples):

```
-H 5C3F0 4BC6
   ^ Error
```

```
      1          1          1
    3A7        F451         C
  + 92A      + CB03       + D
  -----      ------       ---
    CD1       1BF54        19


        1111        1 1
         BCD8        BCD8
       + FAE9      + 0509
       ------      ------
        1B7C1        C1E1
```

Figure 1-4: More examples of hexadecimal addition.

That is an unexpected response. Why does Debug say that you have an error here? The reason has to do with a unit of storage called the *word.* Debug's Hexarithmetic command works only with words, and words happen to be long enough to hold four hex digits, no more.

You will learn more about words in a few pages, but for now, remember that you can work only with four hex digits. Thus, if you try to add two four-digit hex numbers, such as C000h and D000h (which should give you 19000h), you get 9000h instead. Debug keeps only the four rightmost digits of the answer.

```
-H C000 D000
9000  F000
-
```

Converting Decimal to Hex

So far you have only seen the conversion from hex to decimal. Now let's look at how to convert decimal numbers to hex. As mentioned earlier, in Chapter 10 you will create a program to write the 80x86's numbers as decimal numbers; in Chapter 23, you will write another program to read decimal numbers into the 80x86. But, as with decimal-to-hex conversions, you will begin by doing the conversions by hand. Again, start by recalling a bit of grade school math.

When you first learned division, you would divide 9 by 2 to get 4 with a remainder of 1. We will use the remainder to convert decimal numbers to hex. See what happens when you repeatedly divide a decimal number, in this case 493, by 10?

```
493 / 10 = 49 remainder 3 ────────┐
            │                     │
  ┌─────────┘                     │
 49 / 10 = 4 remainder 9 ──────┐  │
            │                  │  │
  ┌─────────┘                  │  │
  4 / 10 = 0 remainder 4 ───┐  │  │
                            │  │  │
                            4  9  3
```

The digits of 493 appear as the remainder in reverse order—that is, starting with the rightmost digit (3). You saw in the last section that all you needed for a hex-to-decimal conversion was to replace powers of 10 with powers of 16. For a decimal-to-hex conversion, can you divide by 16 instead of 10? Indeed, that is the conversion method. For example, find the hex number for 493. You can divide by 16, you get the following:

```
493 / 16 = 30 remainder 13 (Dh)
30 / 16 = 1 remainder 14 (Eh)
1 / 16 = 0 remainder 1 (1h)
4 9 3          =          1 E D h
```

1EDh is the hex equivalent of decimal 493. In other words, keep dividing by 16, and form the final hex number from the remainders. That's all there is to it. Figure 1-5 gives more examples of decimal to hex conversions.

```
1069 / 16 = 66  Remainder  13
  66 / 16 =  4  Remainder   2
   4 / 16 =  0  Remainder   4
1069        =                  4 2 D h
```

```
57,109 / 16 = 3,569  Remainder   5
 3,569 / 16 =   223  Remainder   1
   223 / 16 =    13  Remainder  15
    13 / 16 =     0  Remainder  13
57,109      =                        D F 1 5 h
```

Figure 1-5: More examples of hexadecimal conversions.

Working with Negative Numbers

If you recall, there is still an unanswered puzzle in the number FFFFh. We said that FFFFh is actually –1. Yet, if you convert FFFFh to decimal, we get 65535. How can that be? Does it behave as a negative number?

Well, if you add FFFFh (alias –1) to 5, the result should be 4, because 5 –1 = 4. Is that what happens? Use Debug's H command to add 5 and FFFFh, and you see the following:

```
-H 5 FFFF
0004  0006
-
```

Debug *seems* to treat FFFFh as –1. But FFFFh won't always behave as –1 in programs we will write. To see why not, let's do this addition by hand.

When adding two decimal numbers, you will often find yourself *carrying* a one to the next column, as follows:

```
 1 1
   9 5
+  5 8
------
 1 5 3
```

The addition of two hex numbers isn't much different. Adding 3 to F gives 2, with a carry into the next column, as follows:

```
 1
   F h
+  3 h
------
 1 2 h
```

Now, watch what happens when you add 5 to FFFFh.

```
  1 1 1 1
    0 0 0 5 h
+   F F F F h
  -----------
  1 0 0 0 4 h
```

Since Fh + 1h = 10h, the successive carries neatly move a 1 into the far left position. If you ignore this 1, you have the correct answer for 5 – 1: namely, 4. Strange as it seems, FFFFh behaves as –1 when *you ignore this overflow.* It is called an overflow because the number is now five digits long, but Debug keeps only the last (rightmost) four digits.

Is this overflow an error, or is the answer correct? Well, yes and yes. You can choose either answer. Don't the answers contradict each other? Not really, because you can view these numbers in either of two ways.

Suppose you take FFFFh as equal to 65536. This is a positive number, and it happens to be the largest number you can write with four hex digits. In this case FFFFh is called an *unsigned* number. It is unsigned because you said all four-digit numbers are positive. Adding 5 to FFFFh gives you 10004h; no other answer is correct. In the case of unsigned numbers, then, an overflow is an error.

On the other hand, you can treat FFFFh as a negative number, as Debug did when you used the H command to add FFFFh to 5. FFFFh behaves as –1 as long as you ignore the overflow. In fact, the numbers 8000h through FFFFh all behave as negative numbers. For *signed* numbers, as here, the overflow is not an error.

The 80x86 microprocessor can view numbers either as unsigned or signed; the choice is yours. There are slightly different instructions for each, and we will explore these differences in later chapters as we begin to use numbers on the 80x86. Right now, before you can learn to actually write the negative of, say, 3C8h, we need to unmask the bit and see how it fits into the scheme of bytes, words, and hex.

Bits, Bytes, Words, and Binary Notation

It is time to dig deeper into the intricacies of your PC—time to learn about the arithmetic of the 80x86: binary numbers. The 80x86 microprocessor, with all its power, is rather dumb. It knows only the two digits 0 and 1, so any number it uses must be formed from a long string of 0s and 1s. This is the *binary* (base 2) number system.

When Debug prints a number in hex, it uses a small program to convert its internal numbers from binary to hexadecimal. In Chapter 5, we will build such a program to write binary numbers in hex notation, but first we need to learn more about binary numbers themselves.

Let's take the binary number 1011b (the b stands for binary). This number is equal to the decimal 11, or Bh in hex. To see why, multiply the digits of 1011b by the number's base, 2:

Powers of 2:

$2^3 = 8$
$2^2 = 4$
$2^1 = 2$
$2^0 = 1$

So that:

```
 1 * 8   =   8
 0 * 4   =   0
 1 * 2   =   2
 1 * 1   =   1
_____________
1011b   =  11   or   Bh
```

Likewise, 1111b is Fh, or 15. 1111b is the largest unsigned four-digit binary number you can write, while 0000b is the smallest. Thus, with four binary digits you can write 16 different numbers (see Figure 1-6). There are exactly 16 hex digits, so you can write one hex digit for every four binary digits.

Binary	Decimal	Hexadecimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

Figure 1-6: Binary, hex, decimal for 0 through F.

A two-digit hex number, such as 4Ch, can be written as 0100 1100b. It's comprised of eight digits, which we separate into groups of four for easy reading. Each one of these binary digits is known as a bit, so a number like 0100 1100b, or 4Ch, is eight bits long.

Very often, we find it convenient to number each of the bits in a long string, with bit 0 farthest to the right. The 1 in 10b then is bit number 1, and the leftmost bit in 1011b is bit number 3. Numbering bits in this way makes it easier for us to talk about any particular one, as we will want to later on.

A group of eight binary digits is known as a *byte*, while a group of 16 binary digits, or two bytes, is a *word*, see Figure 1-7. We will use these terms frequently throughout this book, because bits, bytes, and words are all fundamental to the 80x86.

Now you can see why hexadecimal notation is convenient. Two hex digits fit exactly into one byte (four bits per hex digit), and four digits fit exactly into one word. The same cannot be said for decimal numbers. If you try to use two decimal digits for one byte, numbers larger than 99 cannot be written, so you lose the values from 100 to 255—more than half the range of numbers a byte can hold. If you use three decimal digits you must *ignore* more than half the three-digit decimal numbers because the numbers 256 through 999 cannot be contained in one byte.

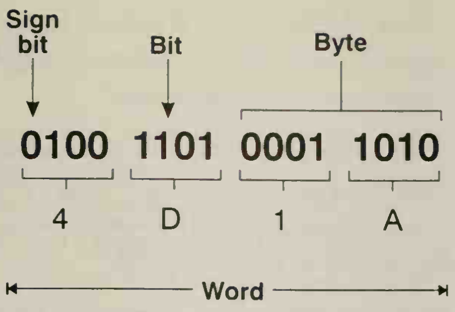


Figure 1-7: A word is made out of bits and bytes.

Two's Complement—An Odd Sort of Negative Number

Now you are ready to learn more about negative numbers. We said before that the numbers 8000h through FFFFh all behave as negative numbers when you ignore the overflow. There is an easy way to spot negative numbers when you write them in binary. The binary forms for positive and negative numbers are as follows:

```
Positive numbers:
    0000h  0000 0000 0000 0000b
      .      .
      .      .
      .      .
    7FFFh  0111 1111 1111 1111b

Negative numbers:
    8000h  1000 0000 0000 0000b
      .      .
      .      .
      .      .
    FFFFh  1111 1111 1111 1111b
```

In the binary forms for all the positive numbers, the leftmost bit (bit 15) is always 0. For all negative numbers, this leftmost bit is always 1. This difference is, in fact, the way that the 80x86 microprocessor knows when a number is negative: It looks at bit 15, the *sign bit.* If you use instructions for unsigned numbers in your programs, the 80x86 will ignore the sign bit, and you will be free to use signed numbers at your convenience.

These negative numbers are known as the *Two's Complement* of positive numbers. Why complement? Because the conversion from a positive number, such as 3C8h, to its two's-complement form is a two-step process, with the first being the conversion of the number to its *complement.*

We won't need to negate numbers often, but we will do the conversion here just so you can see how the 80x86 microprocessor negates numbers. The conversion will seem a bit strange. You won't see why it works, but you will see that it does work.

To find the two's-complement form (negative) of any number, first write the number in binary, ignoring the sign. For example, 4Ch becomes 0000 0000 0100 1100b.

To negate this number, first reverse all the zeros and ones. This process of reversing is called *complementing,* and taking the complement of 4Ch, we find that:

```
0 0 0 0  0 0 0 0  0 1 0 0  1 1 0 0
```

becomes:

```
1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1
```

In the second step of the conversion, add 1:

```
                        1 1
 1 1 1 1  1 1 1 1  1 0 1 1  0 0 1 1
+                                  1
-------------------------------------
 1 1 1 1  1 1 1 1  1 0 1 1  0 1 0 0
                                -4Ch = FFB4h
```

The answer, FFB4h, is the result you get if you use Debug's H command to subtract 4Ch from 0h.

If you wish, you can add FFB4h to 4Ch by hand, to verify that the answer is 10000h. And from our earlier discussion, you know that you should ignore this leftmost 1 to get 0 (4C + (–4C) = 0) when you do two's-complement addition.

Summary

This chapter has been a fairly steep climb into the world of hexadecimal and binary numbers, and it may have required a fair amount of mental exercise. Soon, in Chapter 3, we will slow down to a gentler pace—once you have learned enough to converse with Debug in hex. Now take a breath of fresh air and look back on where you have been and what you have found.

We started out by introducing Debug. In chapters to come, you will become intimate friends with Debug but, since it does not understand your familiar decimal numbers, you began the friendship by learning a new numbering system, hexadecimal notation.

In learning about hex numbers, you also learned how to convert decimal numbers to hex, and hex numbers to decimal. You will write a program to do these translations later, but first you had to learn the language yourself.

Once we covered the basics of hexadecimal notation, we were able to wander off for a look at bits, bytes, words, and binary numbers—important characters you will encounter frequently as you continue to explore the world of the 80x86 and assembly-language programming.

Finally, we moved on to learn about negative numbers in hex—the two's-complement numbers. They led you to signed and unsigned numbers, where you also witnessed overflows of two different types: one in which an overflow leaves the correct answer (addition of two signed numbers), and one in which the overflow leads to the wrong answer (addition of two unsigned numbers).

All this learning will pay off in later chapters, because you will use your knowledge of hex numbers to speak with Debug. Debug will act as an interpreter between you and the 80x86 microprocessor waiting inside your PC.

In the next chapter, you use the knowledge you have gained so far to learn about the 80x86. We will rely on Debug again, and the use of hex numbers, rather than binary, to talk to the 80x86. You will learn about the microprocessor's registers—the places where it stores numbers—and, in Chapter 3, you will be ready to write a real program that will print a character on the screen.

You will also learn more about how the 80x86 does it's math. By the time you reach Chapter 10, you will be able to write a program to convert binary numbers to decimal.

CHAPTER

Doing Arithmetic with the 80x86

In this chapter you will learn how to build your first programs. These programs will be one-line programs that add, subtract, multiply, and divide two numbers. These programs use instructions inside the 80x86 microprocessor.

2

Topics Covered

Using Registers as Variables

Using Memory in the 80x86

Addition, 80x86 Style

Subtraction, 80x86 Style

Negative Numbers in the 80x86

Bytes in the 80x86

Multiplication and Division, 80x86 Style

Summary

Knowing something of Debug's hex arithmetic and the 80x86's binary arithmetic, you can begin to learn how the 80x86 does its math. It uses internal commands called *instructions.*

Using Registers as Variables

Debug, your guide and interpreter, knows much about the 80x86 microprocessor inside your PC. We use it to delve into the inner workings of the 80x86. You begin by asking Debug to display what it can about small pieces of memory called *registers*, in which you can store numbers. Registers are like variables in BASIC, but they are not exactly the same. Unlike the BASIC language, the 80x86 microprocessor contains a fixed number of registers, and these registers are not part of your PC's memory. Ask Debug to display the 80x86's registers with the R, for *Register*, command as follows:

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
-
```

You will probably see different numbers in the second and third lines of your display. Those numbers reflect the amount of memory in a computer. You will continue to see such differences, and later you will learn more about them.

For now, Debug has certainly given you a lot of information. Concentrate on the first four registers, AX, BX, CX, and DX, all of which Debug says are equal to 0000, both here and on your display. These registers are the *general-purpose* registers. The other registers, SP, BP, SI, DI, DS, ES, SS, CS, and IP, are special-purpose registers that we will deal with in later chapters.

The four-digit number following each register name is in hex notation. In Chapter 1, you learned that one word is described exactly by four hex digits. Here, you can see that each of the 13 registers in the 80x86 is one word, or 16 bits long. This is why computers based on the 80x86 microprocessor are known as 16-bit machines. (The 80386 and above microprocessors also have a 32-bit mode, where the registers can be 32 bits long, but currently only Windows NT really uses the 32-bit mode of these processors.)

We mentioned that the registers are like BASIC variables. That means you should be able to change them, and you can. Debug's R command does more than display registers. Followed by the name of the register, the command tells Debug that you wish to view the register and then change it. For example, you can change the AX register as follows:

```
-R AX
AX 0000
:3A7
-
```

Look at the registers again to see if the AX register now contains 3A7h.

```
-R
AX=03A7  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3757  ES=3756  SS=3756  CS=3756  IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
-
```

It does. Furthermore, you can put any hex number into any register with the R command by specifying the register's name and entering the new number after the colon. From here on, we will be using this command whenever you need to place numbers into the 80x86's registers.

You may recall seeing the number 3A7h in Chapter 1, where you used Debug's Hexarithmetic command to add 3A7h and 1EDh. Back then, Debug did the work for you. This time, we will use Debug merely as an interpreter so we can work directly with the 80x86. We will give the 80x86 instructions to add numbers from two registers: we will place a number in the BX register and then instruct the 80x86 to add the number in BX to the number in AX and put the answer back into AX. First, you need a number in the BX register. This time, let's add 3A7h and 92Ah. Use the R command to store 92Ah into BX.

Using Memory in the 80x86

The AX and BX registers should, respectively, contain 3A7h and 92Ah, as you can verify with the R command:

```
AX=03A7  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 E485          IN      AL,85
```

Now that you have these two numbers in the AX and BX registers, how do you tell the 80x86 to add BX to AX? By putting some numbers into your computer's memory.

Your IBM PC probably has at least 640K of memory—far more than you will need to use here. We will place two bytes of *machine code* into a corner of this vast amount of memory. In this case, the machine code will be two binary numbers that tell the 80x86 to add the BX register to AX. Then, we will *execute* this instruction with the help of Debug.

Where should you place the two-byte instruction in memory? And how will you tell the 80x86 where to find it? As it turns out, the 80x86 chops memory into 64K pieces known as *segments.* Most of the time, you will be looking at memory within one of these segments without really knowing where the segment starts. You can do this because of the way the 80x86 labels memory.

All bytes in memory are labeled with numbers, starting with 0h and working up. Remember the four-digit limitation on hex numbers? That means the highest number the 80x86 can use is the hex equivalent of 65535, which means the maximum number of labels it can use is 64K. Even so, experience tells you that the 80x86 can call on more than 64K of memory by being a little bit tricky. It uses two numbers, one for each 64K segment and one for each byte, or *offset,* within the segment. Each segment begins at a multiple of 16 bytes, so by overlapping segments and offsets, the 80x86 effectively can label more than 64K of memory. In fact, this is precisely how the 80x86 uses up to one million bytes of memory. (As you will see in a later chapter, a simple variation on this scheme allows the 80286 and above microprocessor to access more than a single megabyte of memory.)

The addresses (labels) you will use are offsets from the start of a segment. You will write addresses as a segment number, followed by the offset within the segment. For example, 3756:0100 will mean that you are at an offset of 100h within segment 3756h, see Figure 2-1.

In Chapter 11 you will learn more about segments and why the segment number is so high. For now, you can trust Debug to look after the segments for you so that you can work within one segment without having to pay attention to segment numbers. For the time being, we will refer to addresses only by their offsets. Each of these addresses refers to one byte in the segment, and the addresses are sequential, so 101h is the byte following 100h in memory.

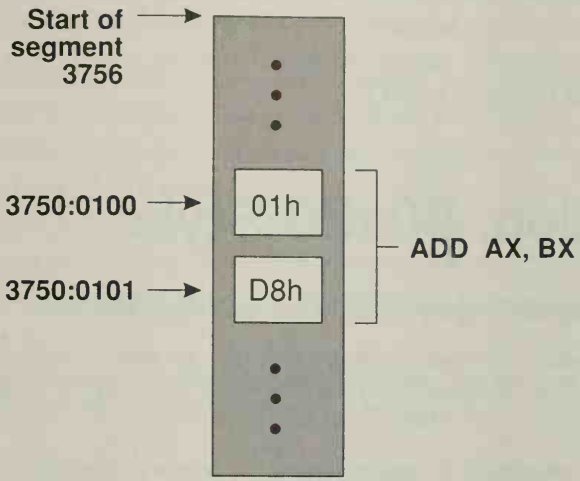


Figure 2-1: Instruction begins 100h bytes from the start of the segment.

Written out, the two-byte instruction to add BX to AX looks like this: ADD AX,BX. We will place this instruction at locations 100h and 101h, in whatever segment Debug starts to use. In referring to your ADD instruction, we will say that it is *at* location 100h, since this is the location of the first byte of the instruction.

Debug's command for examining and changing memory is called E, for *Enter*. Use the following command to enter the two bytes of the ADD instruction.

```
-E 100
3756:0100  E4.01
-E 101
3756:0101  85.D8
-
```

The numbers 01h and D8h are the 80x86's machine language for the ADD instruction at memory locations 3756:0100 and 3756:0101. The segment number you see will probably be different, but that difference won't affect your program. Likewise, Debug probably displayed a different two-digit number for each of your E commands. These numbers (E4h and 85h in our example) are the old numbers in memory at offset addresses 100h and 101h of the segment Debug chose—that is, the numbers are data from previous programs left in memory when you started Debug. (If you just started your computer, the numbers will probably be 00.)

Addition, 80x86 Style

Now your register display should look something like this:

```
AX=03A7  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100   NV UP DI PL NZ NA PO NC
3756:0100 01D8          ADD     AX,BX
```

The ADD instruction is neatly placed in memory, just where you want it to be. You know this from reading the third line of the display. The first two numbers, 3756:0100, give you the address (100h) for the first number of the ADD instruction. Next to this, you see the two bytes for ADD: 01D8. The byte equal to 01h is at address 100h, while D8h is at 101h. Finally, since you entered your instruction in *machine language*—numbers that have no meaning to us, but the 80x86 will interpret as an add instruction—the message ADD AX,BX confirms that you entered the instruction correctly.

Even though you placed the ADD instruction in memory, you are not quite ready to run it through the 80x86 (*execute* it). First, you need to tell the 80x86 where to find the instruction.

The 80x86 finds segment and offset addresses for instructions in two special registers, CS and IP, which you can see listed in the preceding register display. The segment number is stored in the CS, or *Code Segment*, register, which we will discuss shortly. If you look at the register display, you can see that Debug has already set the CS register for you (CS=3756, in our example). The full starting address of your instruction, however, is 3756:0100.

The second part of this address (the offset within segment 3756) is stored in the IP (*Instruction Pointer)* register. The 80x86 uses the offset in the IP register to actually find our first instruction. You can tell it where to look by setting the IP register to the address of your first instruction—IP=0100.

The IP register is already set to 100h because Debug sets IP to 100h whenever you first start it. Knowing this, we have deliberately chosen 100h as the address of the first instruction and have thus eliminated the need to set the IP register in a separate step. It is a good point to keep in mind.

Now, with the instruction in place and the registers set correctly, we will tell Debug to execute our one instruction. We will use Debug's T (*Trace*) command, which executes one instruction at a time and then displays the registers. After each trace, the IP should point to the next instruction. In this case, it will point to 102h. Since we have not put an instruction at 102h, the last line of the register display will show an instruction left over from some other program. Ask Debug to trace one instruction with the T command as follows:

```
-T
AX=0CD1  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0102   NV UP DI PL NZ AC PE NC
3756:0102 AC            LODSB
-
```

The AX register now contains CD1h, which is the sum of 3A7h and 92Ah. The IP register points to address 102h, so the last line of the register display shows some instruction at memory location 102h, rather than 100h. Figures 2-2 and 2-3 summarize the before and after of running the add instruction.

We mentioned earlier that the instruction pointer, together with the CS register, always points to the next instruction for the 80x86. If you typed T again, you would execute the next instruction, but don't do it just yet—your 80x86 might head for limbo.

Instead, what if you want to execute the ADD instruction again, adding 92Ah to CD1h and storing the new answer in AX? For that you need to tell the 80x86 where to find its next instruction, which you want to be the ADD instruction at 0100h. Can you just change the IP register to 0100h? Try it by using the R command to set IP to 100, and look at the register display as follows:

```
AX=0CD1  BX=092A  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3756  ES=3756  SS=3756  CS=3756  IP=0100   NV UP DI PL NZ AC PE NC
3756:0100               ADD     AX,BX
```

That's done it. Try the T command again and see if the AX register contains 15FBh. It does.

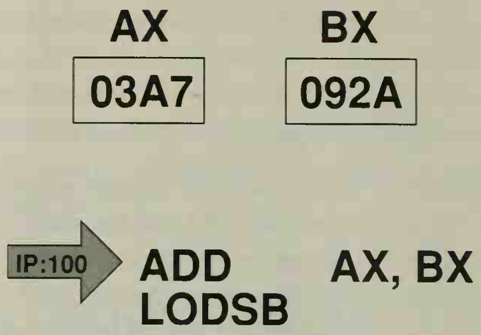


Figure 2-2: Before executing the ADD instruction.

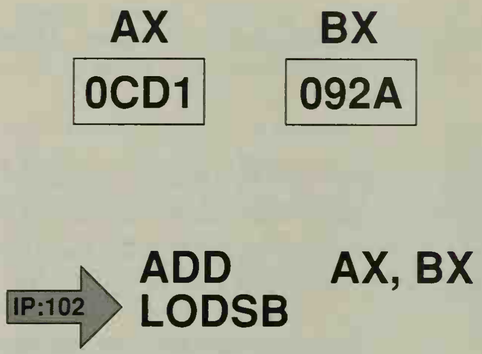


Figure 2-3: After executing the ADD instruction.

You should always check the IP register and the instruction at the bottom of an R display before using the T command. That way, you will be sure the 80x86 executes the instruction you want it to.

Now, set the IP register to 100h once again and make certain that the registers contain AX = 15FB, BX = 092A. Now you will try subtraction.

Subtraction, 80x86 Style

We are going to write an instruction that will subtract BX from AX. After two subtractions, you will have 3A7h in AX—the point from which you started before your two additions. You will also see an easier way to enter two bytes into memory.

When you entered the two bytes for your ADD instruction, you typed the E command twice: once with 0100h for the first address, and once with 0101h for the second address. The procedure worked, but as it turns out you can actually enter the second byte without another E command if you separate it from the first byte with a space. When you have finished entering bytes, pressing the Enter key will exit from the Enter command. Try this method for your subtract instruction:

```
-E 100
3756:0100  01.29   D8.D8
```

The register display (remember to reset the IP register to 100h) should now show the instruction SUB AX,BX, which subtracts the BX register from the AX register and leaves the result in AX. The order of AX and BX may seem backwards, but the instruction is like the BASIC statement AX = AX – BX except that the 80x86, unlike BASIC, always puts the answer into the first variable (register).

Execute this instruction with the T command. AX should contain CD1. Change IP to point back to this instruction, and execute it again (remember to check the instruction at the bottom of the R display first). AX should now be 03A7.

Negative Numbers in the 80x86

In the last chapter, you learned how the 80x86 uses the two's-complement form for negative numbers. Now, you will work directly with the SUB instruction to calculate negative numbers. Put the 80x86 to a test, to see whether you get FFFFh for –1. If you subtract one from zero and, if correct, the subtraction should place FFFFh (–1) into AX. Set AX equal to zero and BX to one, then trace through the instruction at address 0100h. Just what you expected: AX = FFFFh.

While you have this subtraction instruction handy, you may wish to try some different numbers to gain a better feel for two's-complement arithmetic. For example, see what result you get for –2.

Bytes in the 80x86

All of your arithmetic so far has been performed on words, hence the four hex digits. Does the 80x86 microprocessor know how to perform math with bytes? Yes, it does.

Since one word is formed from two bytes, each general-purpose register can be divided into two bytes, known as the *high byte* (the first two hex digits) and the *low byte* (the second two hex digits). Each of these registers can be called by its letter (A through D), followed by X for a word, H for the high byte, or L for the low byte. For example, DL and DH are byte registers, and DX is a word register, see Figure 2-4.

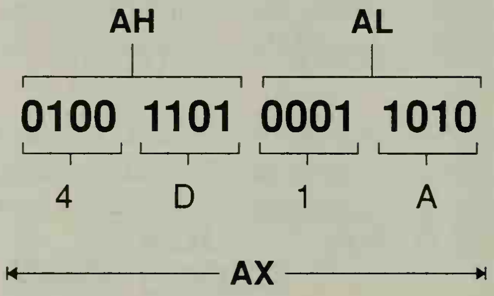


Figure 2-4: The AX register split into two byte registers (AH and AL).

To test byte-sized math with an ADD instruction, you enter the two bytes 00h and C4h, starting at location 0100h. At the bottom of the register display, you will see the instruction ADD AH,AL, which will add the two bytes of the AX register and place the result in the high byte, AH.

Next, load the AX register with 0102h. This places 01h in the AH register and 02h in the AL register. Set the IP register to 100h, execute the T command, and you will find that AX now contains 0302. The result of 01h + 02h is 03h, and that value is in the AH register.

Suppose you had not meant to add 01h and 02h. Suppose you really meant to add 01h and 03h. If the AX register already contained 0102, could you use Debug to change the AL register to 03h? No. You have to change the entire

AX register to 0103h because Debug only allows us to change *word* registers. There isn't a way to change just the low or high part of a register with Debug. But, as you saw in the last chapter, this isn't a problem. With hex numbers, a word can be split into two bytes by breaking the four-digit hex number in half. Thus, the word 0103h becomes the two bytes 01h and 03h.

To try this ADD instruction, load the AX register with 0103h. Your ADD AH,AL instruction is still at memory location 0100h, so reset the IP register to 100h and, with 01h and 03h now in the AH and AL registers, trace through this instruction. This time, AX contains 0403h: 04h, the sum of 01h + 03h is now in the AH register.

Multiplication and Division, 80x86 Style

You have seen the 80x86 add and subtract two numbers. Now you will see that it can also multiply and divide—clever processor. The multiply instruction is called MUL, and the machine code to multiply AX and BX is F7h E3h. We will enter this into memory, but first a word about the MUL instruction.

Where does the MUL instruction store its answer? In the AX register? Not quite; we have to be careful here. As you will soon see, multiplying two 16-bit numbers can give a 32-bit answer, so the MUL instruction stores its result in two registers, DX and AX. The higher 16 bits are placed in the DX register; the lower, into AX. We will write this register combination as DX:AX, from time to time.

Back to Debug and the 80x86. Enter the multiply instruction, F7h E3h, at location 0100h, just as you did for the addition and subtraction instructions, and set AX = 7C4Bh and BX = 100h. You will see the instruction in the register display as MUL BX, without any reference to the AX register. To multiply words, as here, the 80x86 always multiplies the register you name in the instruction by the AX register, and stores the answer in the DX:AX pair of registers.

Before you actually execute this MUL instruction, do the multiplication by hand. The three digits 100 have the same effect in hex as in decimal, so to multiply by 100h simply add two zeros to the right of a hex number. Thus, 100h * 7C4Bh = 7C4B00h. This result is too long to fit into one word, so we will split it into the two words 007Ch and 4B00h.

Use Debug to trace through the instruction. You will see that DX contains the word 007Ch, and AX contains the word 4B00h. In other words, the 80x86 returned the result of the *word-multiply* instruction in the DX:AX pair of registers (see Figures 2-5 and 2-6). Since multiplying two words together can never be longer than two words, but will often be longer than one word (as you just saw), the word-multiply instruction *always* returns the result in the DX:AX pair of registers.

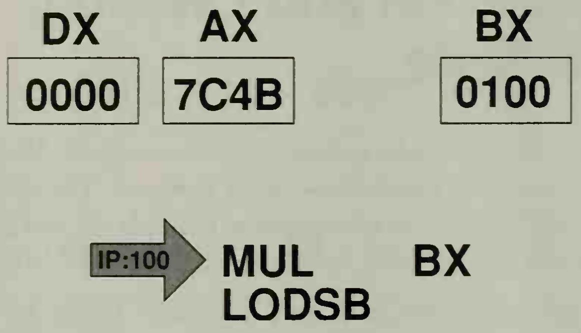


Figure 2-5: Before executing the MUL instruction.

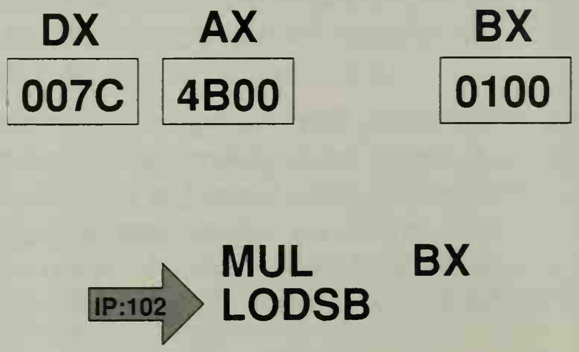


Figure 2-6: After executing the MUL instruction.

What about division? When you divide numbers are divided, the 80x86 keeps both the result and the remainder of the division. To see how the 80x86's division works, first place the instruction F7h F3h at 0100h (and 101h). Like the MUL instruction, DIV uses DX:AX without being told, so all you see is DIV BX. Now, load the registers so that DX = 007Ch and AX = 4B12h; BX should still contain 0100h.

Again, let's first calculate results by hand: 7C4B12h / 100h = 7C4Bh, with 12h left over. When you execute the division instruction at 0100h, you find that AX = 7C4Bh, the result of your division; and DX = 0012h, which is the remainder (see Figures 2-7 and 2-8). (We will put this remainder to use in Chapter 10, when we write a program to convert decimal numbers to hex by using the remainders, just as in Chapter 1.)

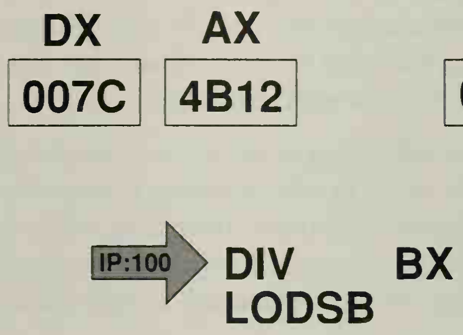


Figure 2-7: Before executing the DIV instruction.

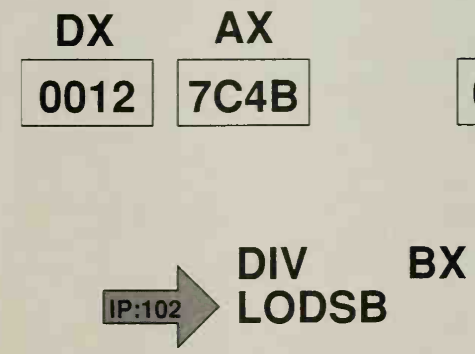


Figure 2-8: After executing the DIV instruction.

Summary

It is almost time to write a real program—one to print a character on the screen. You have put in your time learning the basics. First take a look at the ground that you have covered, and then we will move on.

You began this chapter by learning about registers and noticing their similarity to variables in BASIC. Unlike BASIC, however, the 80x86 has a small, fixed number of registers. We concentrated on the four general-purpose registers (AX, BX, CX, and DX), with a quick look at the CS and IP registers, which the 80x86 uses to locate segment and offset addresses.

After learning how to change and read registers, we moved on to build some single-instruction programs by entering the machine codes to add, subtract, multiply, and divide two numbers with the AX and BX registers. In future chapters we will use much of what you learned here, but you won't need to remember the machine codes for each instruction.

You also learned how to tell Debug to execute or trace through a single instruction. You will come to rely heavily on Debug to trace through your programs. Of course, as programs grow in size, this tracing will become both more useful and more tedious. Later on you will build on your experience and learn how to execute more than one instruction with a single Debug command.

Let's turn back to real programs and learn how to make a program that speaks.

CHAPTER

Printing Characters

In this chapter you will build several real programs. You will start with a one-line program that displays a single character on your screen. Then you will turn this into a two-line program that displays a character and stops running, all by itself. Finally, you will build a program that displays a whole string of characters on your screen.

3

Topics Covered

You now know enough to do something solid, so roll up your sleeves and flex your fingers. We will begin by instructing DOS to send a character to the screen. Then move on to even more interesting work—building a small program with more than one instruction, and from there, learning another way to put data into registers from within a program. Now, onto getting DOS to speak.

INT—Using DOS Functions

We will add a new instruction, called INT (for *Interrupt*) to your four math instructions ADD, SUB, MUL, and DIV. INT is something like a subroutine call in any other programming language. We will use the INT instruction to ask DOS to print a character, A, on the screen for us.

Before you learn how INT works, let's run through an example. Start Debug and place 200h into AX and 41h into DX. The INT instruction for DOS functions is INT 21h—in machine code, CDh 21h. This is a two-byte instruction like the DIV instruction in the last chapter. Put INT 21h in memory, starting at location 100h, and use the R command to confirm that the instruction reads INT 21 (remember to set IP to 100h if it isn't already there).

Now you are ready to execute this instruction, but you cannot use the trace command here as you did in the last chapter. The trace command executes one instruction at a time, but the INT instruction calls upon a large program in DOS to do the actual work.

You don't want to execute each of the instructions in the entire DOS "subroutine" by tracing through it one instruction at a time. Instead, you want to *run* your one-line program, but stop before executing the instruction at location 102h. You can do this with Debug's G (*Go till*) command, followed by the address at which you want to stop. The command is as follows:

```
-G 102
A
AX=0241  BX=0000  CX=0000  DX=0041  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3970  IP=0102   NV UP DI PL NZ NA PO NC
3970:0102 8BE5          MOV     SP,BP
```

DOS printed the character A, and then returned control to your small program. (Remember, the instruction at 102h is just data left behind by another program, so you will probably see something different.) Your small program is, in a sense, two instructions long, the second instruction being whatever is at location 102h. It looks something like the following:

```
INT     21
MOV     SP,BP          (Or whatever is on your computer)
```

This random second instruction will soon be replaced with one of your own. For now, since it isn't anything you want to execute, you told Debug to run your program, stop execution when it reached the second instruction, and display the registers when it was done.

How did DOS know that it should print the A? The 02h in the AH register told DOS to print a character. Another number in AH would tell DOS to execute a different function. (You will see others later, but if you are curious right now, you can find a list of functions in Appendix D that will be used in this book.)

As for the character, DOS uses the number in the DL register as the ASCII code for the character to print when you ask it to send a character to the screen. You stored 41h, the ASCII code for an uppercase A.

In Appendix D, you will find a chart of ASCII character codes for all the characters your PC can display. The numbers are in both decimal and hex notation. Since Debug can only read hex, here is a good chance for you to practice converting decimal numbers to hex. Pick a character from the table and convert it to hex on your own. Then, verify your conversion by typing your hex value into the DL register and running the INT instruction again (remember to reset IP to 100h).

You may have wondered what would have happened if you had tried the trace command on the INT instruction. Suppose that you had not executed the G 102 command and, instead, traced a short distance through it to see what happens. If you try this yourself, don't go too far—you may find your PC doing something strange. After you have traced through a few steps, exit Debug with the Q command. This will clean up any mess you have left behind.

```
-R

AX=0200  BX=0000  CX=0000  DX=0041  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3970  IP=0100   NV UP DI PL NZ NA PO NC
3970:0100 CD21          INT     21
-T

AX=0200  BX=0000  CX=0000  DX=0041  SP=FFE8  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3372  IP=0180   NV UP DI PL NZ NA PO NC
3372:0180 80FC4B        CMP     AH,4B
-T

AX=0200  BX=0000  CX=0000  DX=0041  SP=FFE8  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3372  IP=0183   NV UP DI NG NZ AC PE CY
3372:0183 7405          JZ      018A
-T

AX=0200  BX=0000  CX=0000  DX=0041  SP=FFE8  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3372  IP=0185   NV UP DI NG NZ AC PE CY
3372:0185 2E            CS:
3372:0186 FF2EAB0B      JMP     FAR [0BAB]                  CS:0BAB=0BFF
-Q
```

Notice that the first number of the address changed here, from 3970 to 3372. These last three instructions were part of DOS, and the program for DOS is in another segment. In fact, there are many more instructions that DOS executes before it prints a single character. Even an apparently simple task is not as easy as it sounds. Now you can understand why you used the G command to run the program to location 102h. If you had not used the G command, there would have been a torrent of instructions from DOS. (If you are using a different version of DOS, the instructions you see when you try this may be different.)

Exiting Programs—INT 20h

Do you remember that your INT instruction was 21h? If you changed the 21h to a 20h, you would have INT 20h instead. INT 20h is another interrupt instruction, and it tells DOS we want to exit our program so DOS can take full control again. In our case, INT 20h will send control back to Debug, since we are executing our programs from Debug, rather than from DOS.

Enter the instruction CDh 20h, starting at location 100h, then try the following (remember to check the INT 20h instruction with the R command):

```
-G 102

Program terminated normally
-R

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3970  IP=0100   NV UP DI PL NZ NA PO NC
3970:0100 CD20          INT     20
-G

Program terminated normally
-R

AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3970  ES=3970  SS=3970  CS=3970  IP=0100   NV UP DI PL NZ NA PO NC
3970:0100 CD20          INT     20
```

The command G, with no number after it, executes the entire program (which is just one instruction now, because INT 20 is an *exit* instruction), and then returns to the start. IP has been reset to 100h, which is where we started. The registers in this example are 0 only because we started Debug afresh.

You can use this INT 20h instruction at the end of a program to return control gracefully to DOS (or Debug). Next, you put this instruction together with INT 21h and build a two-line program.

Putting the Pieces Together— A Two-Line Program

Starting at location 100h, enter the two instructions INT 21h, INT 20h (CDh 21h CDh 20h) one after the other. (From now on, we will always start programs at location 100h.)

When you had only one instruction, you could "list" that instruction with the R command, but now you have two instructions. To see them, use the U (*Unassemble*) command, which lists part of a program in memory:

```
-U 100

3970:0100 CD21          INT     21
3970:0102 CD20          INT     20
3970:0104 D98D460250B8  ESC     09,[DI+0246][DI+B850]
3970:010A 8D00          LEA     AX,[BX+SI]
3970:010C 50            PUSH    AX
3970:010D E82A23        CALL    243A
3970:0110 8BE5          MOV     SP,BP
3970:0112 83C41A        ADD     SP,+1A
3970:0115 5D            POP     BP
3970:0116 C3            RET
3970:0117 55            PUSH    BP
3970:0118 83EC02        SUB     SP,+02
3970:011B 8BEC          MOV     BP,SP
3970:011D 823E0E0000    CMP     BYTE PTR [000E],00
```

The first two instructions are recognized as the two instructions which were just entered. The other instructions are remnants left in memory. As your program grows, you will fill this display with more of your own code.

Now, fill the AH register with 02h and the DL register with the number for any character (just as you did earlier when you changed the AX and DX registers), then simply type the G command to see your character. For example, if you place 41h into DL, you see the following:

```
-G

A
Program terminated normally
-
```

Try this a few times with other characters in DL before we move on to other ways to set these registers.

Entering Programs

From here on, most of our programs will be more than one instruction long. To present these programs, we will use an unassemble display. The last program would appear as follows:

```
3970:0100 CD21          INT     21
3970:0102 CD20          INT     20
```

So far, you have entered the instructions for your programs directly as numbers, such as CDh, 21h. But that is a lot of work, and, as it turns out, there is a much easier way to enter instructions.

In addition to the unassemble command, Debug includes an A (*Assemble*) command, which allows you to enter the mnemonic, or human-readable, instructions directly. So rather than entering those cryptic numbers for your short program, you can use the assemble command to enter the following:

```
-A 100
3970:0100 INT 21
3970:0102 INT 20
3970:0104
-
```

When you have finished assembling instructions, all you have to do is press the Enter key, and the Debug prompt reappears. Here, the A command told Debug that you wished to enter instructions in mnemonic form, and the 100 told Debug to start entering instructions at location 100h. Since Debug's assemble command makes entering programs much simpler, we will use it from now on to enter instructions.

MOVing Data into Registers

Although we have relied on Debug quite a bit, we won't always run programs with it. Normally, a program would set the AH and DL registers itself before an INT 21h instruction. To do this, you will learn another instruction, called MOV. Once you know enough about this instruction, you can take your small program to print a character and make a real program—one that can be executed directly from DOS.

Soon, the MOV instruction will be used to load numbers into registers AH and DL. But let's start learning about MOV by moving numbers between registers. Place 1234h into AX (12h into the AH register, and 34h in AL) and ABCDh into DX (ABh in DH, and CDh in DL). Now, enter the following instruction with the A command:

```
396F:0100 88D4          MOV     AH,DL
```

This instruction *moves* the number in DL into AH by putting a copy of it into AH; DL is not affected. If you trace through this one line, you will find that AX = CD34h and DX = ABCDh. Only AH has changed. It now holds a copy of the number in DL.

Like the BASIC assignment statement AH = DL, a MOV instruction copies a number from the second register to the first, and for this reason we write AH before DL. Although there are some restrictions, which you will find out about later, you can use other forms of the MOV instruction to copy numbers between other pairs of registers. For example, reset IP and try the following:

```
396F:0100 89C3          MOV     BX,AX
```

You have just moved words, rather than bytes, between registers. The MOV instruction always works between words and words, or bytes and bytes; never between words and bytes. It makes sense, for how would you move a word into a byte?

We originally set out to move a number into the AH and DL registers. Let's do so now with another form of the MOV instruction, which is as follows:

```
396F:0100 B402          MOV     AH,02
```

This instruction moves 02h into the AH register without affecting the AL register. The second byte of the instruction, 02h, is the number we wish to move. Try moving a different number into AH. Change the second byte to another number, such as C1h, with the E 101 command.

Now, put all the pieces of this chapter together and build a longer program. This one will print an asterisk, *, all by itself, with no need to set the AH and DL registers. The program uses MOV instructions to set the AH and DL registers before the INT 21h call to DOS. The instructions are as follows:

```
396F:0100 B402          MOV     AH,02
396F:0102 B22A          MOV     DL,2A
396F:0104 CD21          INT     21
396F:0106 CD20          INT     20
```

Enter the program and check it with the U command (U 100). Make sure IP points to location 100h, then try the G command to run the entire program. You should see the * character appear on your screen, as follows:

```
-G
*
Program terminated normally
-
```

Now that you have a complete, self-contained program, you can write it to disk as a COM program, so you will be able to execute it directly from DOS. We can run a .COM program from DOS simply by typing its name. Since your program doesn't yet have a name, you need to give it one. The Debug command N (*Name*) gives a name to a file before you write it to disk. Type the following command to give the name WRITESTR.COM to our program.

```
-N WRITESTR.COM
```

This command does not write a file to the disk, though—it simply names the file that will be written with another command that you will use below.

Next, you must give Debug a byte count, telling it the number of bytes in the program so it will know how much memory you want to write to your file. If you refer to the unassemble listing of your program, you can see that each instruction is two bytes long (this won't always be true). There are four instructions, so your program is 4 * 2 = 8 bytes long. (You could also put Debug's H command to work and use hexarithmetic to determine the number of bytes in your program. Typing H 108 100 to subtract the first address after your program, 108, from 100 will produce 8.)

Once you have your byte count, you need somewhere to put it. Debug uses the pair of registers BX:CX for the length of your file, so putting 8h into CX tells Debug that your program is eight bytes long. Finally, since your file is only eight bytes long, you also need to set BX to zero. Once the name and length of the program is set, write it to disk with Debug's W (for *Write*) command, as follows:

```
-W
Writing 00008 bytes
-
```

You now have a program on your disk called WRITESTR.COM, so exit Debug with a Q, and look for it. Use the DOS Dir command to list the file:

```
C>DIR WRITESTR.COM

 Volume in drive C has no label
 Directory of  C:\
WRITESTR COM        8 08-28-92  10:05a
        1 File(s)          8 bytes
                     663552 bytes free

C>
```

The directory listing tells you that WRITESTR.COM is on the disk and that it is eight bytes long. To run the program, type *Writestr* at the DOS prompt. You will see a * appear on the display.

Writing a String of Characters

As a final example for this chapter, you use INT 21h, with a different function number in the AH register, to write a whole string of characters. You will have to store the string of characters in memory and tell DOS where to find the string. In the process, you will also learn more about addresses and memory.

You have already seen that function number 02h for INT 21h prints one character on the screen. Another function, number 09h, prints an entire string, and stops printing characters when it finds a $ symbol in the string. To put a string into memory, start at location 200h, so the string won't become tangled with the code for your program. Enter the following numbers by using the instruction E 200:

```
48      65      6C      6C
6F      2C      20      44
4F      53      20      68
65      72      65      2E
24
```

The last number, 24h, is the ASCII code for a $ sign, and it tells DOS that this is the end of your string of characters. You will see what this string says in a minute, when you run the program, which you should enter as follows:

```
396F:0100 B409          MOV     AH,09
396F:0102 BA0002        MOV     DX,0200
396F:0105 CD21          INT     21
396F:0107 CD20          INT     20
```

200h is the address of the string you entered, and loading 200h into the DX register tells DOS where to find the string of characters. Check your program with the U command, then run it with a G command:

```
-G
Hello, DOS here.
Program terminated normally
```

Now that you have stored some characters in memory, it is time to meet another Debug command, D (for *Dump*). The dump command dumps memory to the screen somewhat like U lists instructions. Just as when you use the U command, simply place an address after D to tell Debug where to start the dump. For example, type the command D 200 to see a dump of the string you just entered.

```
-D 200
396F:0200  48 65 6C 6C 6F 2C 20 44-4F 53 20 68 65 72 65 2E   Hello, DOS here.
396F:0210  24 5D C3 55 83 EC 30 8B-EC C7 06 10 00 00 00 E8   $]CU.10.1G.....h
      .
      .
      .
```

After each pair of address numbers (such as 396F:0200 in the example), you see 16 hex bytes, followed by the 16 ASCII characters for these bytes. On the first line you see most of the ASCII codes and characters you typed in. The ending $ sign you typed is the first character on the second line; the remainder of that line is a miscellaneous assortment of characters.

Wherever you see a period (.) in the ASCII window, it represents either a period or a special character, such as the Greek letter pi. Debug's D command displays only 96 of the 256 characters in the PC character set, so a period is used for the remaining 160 characters.

You can use the D command in the future to check numbers entered for data, whether those data are characters or ordinary numbers. (For more information, refer to the Debug section in your DOS manual.)

Your string-writing program is complete, so you can write it to the disk. The procedure is the same one you used to write WRITESTR.COM to disk, except this time you have to set your program length to a value long enough to include the string at 200h. The program begins at line 100h, and you can see from the memory dump just performed that the first character (]) following the $ sign that ends our string is at location 211h. You can use the H command to find the difference between these two numbers. Find 211h – 100h and store this value into the CX register and set BX to zero. Use the N command to give the program a name (with the .COM extension so you can run the program directly from DOS), then use the W command to write the program and data to a disk file.

That's it for writing characters to the screen, aside from one final note—DOS never sends the $ character. This occurs because DOS uses the $ sign to mark the end of a string of characters. That means you cannot use DOS to print a string with a $ in it, but in a later chapter, you will learn how to print a string with a $ sign or any other special character.

Summary

The first two chapters brought you to the point where you could work on a real program. In this chapter, you used your knowledge of hex numbers, Debug, 80x86 instructions, and memory to build short programs to print a character and a string of characters on the screen. In the process you also learned some new things.

First you learned about INT instructions—not in much detail, but enough to write two short programs. In later chapters, you will gain more knowledge about interrupt instructions as you increase your understanding of the 80x86 microprocessor tucked under the cover of your PC.

Debug has, once again, been a useful and faithful guide. We have been relying heavily on Debug to display the contents of registers and memory, and in this chapter we used its abilities even more. Debug ran your short programs with the G command.

You also learned about the INT 20 exit instruction as well as the MOV instruction for moving numbers into and between registers. The exit instruction (INT 20) allowed you to build a complete program that could be written to the disk and run directly from DOS without the help of Debug. The MOV instruction gave you the ability to set registers before an INT 21 (print) instruction, so you could write a self-contained program to print one character.

Finally, we rounded out the chapter with the INT 21h function to print an entire string of characters. We will use all these instructions heavily throughout the rest of this book, but as you saw from using the Debug assemble and unassemble commands, you won't need to remember the machine codes for these instructions.

Now you know enough to move on to printing binary numbers. In the next chapter you will build a short program to take one byte and print it on the screen as a string of binary digits (zeros and ones).

Printing Binary Numbers

In this chapter you will learn several new things that you will use to build a small program that displays a number on your screen in binary notation. The tools you will use are the RCL (rotate carry left) instruction, the carry flag, the ADC (add with carry), and the LOOP instruction.

4

Topics Covered

In this chapter you will build a program to write binary numbers to the screen as strings of zeros and ones. You have most of the knowledge you need, and your work here will help solidify ideas you have already covered. This chapter also adds a few instructions to those you already know, including another version of ADD and some instructions to help repeat parts of your program. Let's begin by learning something completely new.

Rotating Numbers through the Carry Flag

In Chapter 2, when you first encountered hex arithmetic, you found that adding 1 to FFFFh should give you 10000h, but does not. Only the four hex digits to the right fit into one word; the 1 does not fit. You also found that this 1 is an overflow and that it is not lost. Where does it go? It is put into something called a *flag*—in this case, the *Carry Flag,* or *CF.* Flags contain one-bit numbers, so they can hold either a zero or a one. If you need to carry a one into the fifth hex digit, it goes into the carry flag.

Go back to the ADD instruction of Chapter 3 (ADD AX,BX). Put FFFFh into AX and 1 into BX, then trace through the ADD instruction. At the end of the second line of Debug's R display, you will see eight pairs of letters. The last of these, which can read either NC or CY, is the carry flag. Right now, because your ADD instruction resulted in an overflow of 1, you will see that the carry status reads CY (*Carry*). The carry bit is now 1 (or, as we will say, it's set).

To confirm that you have stored a seventeenth bit here (it would be the ninth bit for a byte addition), add one to the zero in AX by resetting IP to 100h and tracing through the ADD instruction again. The carry flag is affected by each ADD instruction, and this time there shouldn't be any overflow, so the carry should be reset. The carry does become zero, as indicated by the NC, which stands for *No Carry,* in the R display.

You learn about other status flags later in this book. If you are curious, you can find information about them right now under Debug's R command in your DOS manual.

Let's review the task of printing a binary number to see how the carry information could be useful. You will print only one character at a time, so you want to pick off the bits of your number, one by one, from left to right. For example, the first character you would want to print in the number 1000 0000b would be the one. If you could move this entire byte left one place, dropping the one into the carry flag and adding a 0 to the right side, then repeat the process for each succeeding digit, the carry flag would pick off your binary digits. You can do just this with a new instruction called RCL (*Rotate Carry Left*). To see how it functions, enter the short program as follows:

```
3985:0100 D0D3          RCL     BL,1
```

This instruction *rotates* the byte in BL to the left by one bit (hence the ,1) through the carry flag. The instruction is called rotate, because RCL moves the leftmost bit into the carry flag, while moving the bit currently in the carry flag into the rightmost bit position (0) as in Figure 4-1. In the process, all the other bits are moved or rotated to the left. After enough rotations (17 for a word, 9 for a byte) the bits are moved back into their original positions and you get back the original number. Place B7h in the BX register, then trace through this rotate instruction several times. Converting your results to binary, you will see the following:

```
Carry         BL register
  0        1 0 1 1   0 1 1 1     B7h       We start here
  1        0 1 1 0   1 1 1 0     6Eh
  0        1 1 0 1   1 1 0 1     DDh
  1        1 0 1 1   1 0 1 0     BAh
                  .
                  .
                  .
  0        1 0 1 1   0 1 1 1     B7h        After 9 rotations
```

In the first rotation, bit 7 of BL moves into the carry flag; the bit in the carry flag moves into bit 0 of BL; all other bits move left one position. Succeeding moves continue rotating the bits to the left until, after nine rotations, the original number is back in the BL register.

You are getting closer to building a program to write binary numbers to the screen, but you still need a few other pieces. Let's see how you can convert the bit in the carry flag into the character 0 or 1.

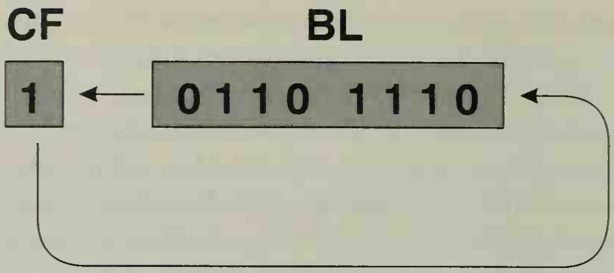


Figure 4-1: The RCL BL,1 instruction.

Adding the Carry Flag to a Number

The normal ADD instruction, for example, ADD AX,BX, simply adds two numbers. Another instruction, ADC (*Add with Carry*), adds three numbers: the two, as before, *plus* one bit from the carry flag. If you look in your ASCII table, you will discover that 30h is the character 0 and 31h is the character 1. So, adding the carry flag to 30h gives the character 0 when the carry is clear, and 1 when the carry is set. Thus, if DL = 0 and the carry flag is set (1), executing the following instruction adds DL(0) and 1h (the carry) to 30h ('0'), which gives 31h ('1').

```
ADC  DL,30
```

With one instruction you have converted the carry to a character that you can print.

At this point, rather than run through an example of ADC, wait until you have a complete program. After you have built the program you will execute its instructions one at a time in a procedure called *single-stepping.* Through this procedure you will see both how the ADC instruction works and how it fits nicely into your program. First, you need one more instruction which you will use to repeat your RCL, ADC, and INT 21h (print) instructions eight times—once for each bit in a byte.

Looping—Repeating a Block of Code

As mentioned, the RCL instruction is not limited to rotating bytes; it can also rotate entire words. We will use this ability to demonstrate the *LOOP* instruction. LOOP is similar to the FOR-NEXT command in Basic, but not as general. As with BASIC's FOR-NEXT loop, you need to tell LOOP how many times to run through a loop. You do this by placing your repeat count in register CX. Each time it goes through the loop, the 80x86 subtracts one from CX; when CX becomes zero, LOOP ends the loop.

Why the CX register? The C in CX stands for *Count*. You can use this register as a general-purpose register, but as you will see in the next chapter, you can also use the CX register with other instructions when you wish to repeat operations.

Here is a simple program that rotates the BX register left eight times, moving BL into BH (but not the reverse, because you rotate through the carry flag):

```
396F:0100 BBC5A3        MOV     BX,A3C5
396F:0103 B90800        MOV     CX,0008
396F:0106 D1D3          RCL     BX,1
396F:0108 E2FC          LOOP    0106
396F:010A CD20          INT     20
```

The loop starts at 106h (RCL BX,1) and ends with the LOOP instruction. The number following LOOP (106h) is the address of the RCL instruction. When you run the program, LOOP subtracts one from CX, then jumps to address 106h if CX is not equal to zero. The instruction RCL BX,1 (Rotate Carry Left, one place) is executed eight times here, because CX is set to eight before the loop.

You may have noticed that, unlike the FOR-NEXT loop in BASIC, the LOOP instruction is at the end of our loop (where you would put the NEXT statement in BASIC). The start of the loop, the RCL instruction at 106h, has no special instruction like FOR has in BASIC. If you know a language like Pascal, you can see that the LOOP instruction is somewhat akin to the REPEAT-UNTIL pair of instructions, where the REPEAT instruction just labels the start of the block of instructions to loop through.

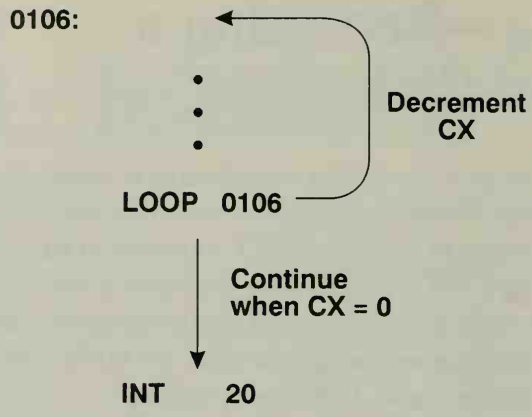


Figure 4-2: The LOOP instruction.

There are different ways you could execute your small program. If you simply type G, you won't see any change in the register display, because Debug saves all the registers before it starts carrying out a G command. Then, if it encounters an INT 20 instruction (as it will in your program), it restores all the registers. Try G. You will see that IP has been reset to 100h (where you started), and that the other registers don't look any different, either.

If you have the patience, you can trace through this program, instead. Taking it one step at a time, you can watch the registers change at each step, as follows:

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0100   NV UP DI PL NZ NA PO NC
0CDE:0100 BBC5A3        MOV     BX,A3C5
-T

AX=0000  BX=A3C5  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0103   NV UP DI PL NZ NA PO NC
0CDE:0103 B90800        MOV     CX,0008
-T

AX=0000  BX=A3C5  CX=0008  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0106   NV UP DI PL NZ NA PO NC
0CDE:0106 D1D3          RCL     BX,1
-T
```

```
AX=0000  BX=478A  CX=0008  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0108   OV UP DI PL NZ NA PO CY
0CDE:0108 E2FC          LOOP    0106
-T
AX=0000  BX=478A  CX=0007  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0106   OV UP DI PL NZ NA PO CY
0CDE:0106 D1D3          RCL     BX,1
                              .
                              .
                              .
-T

AX=0000  BX=C551  CX=0001  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0108   NV UP DI PL NZ NA PO CY
0CDE:0108 E2FC          LOOP    0106
-T

AX=0000  BX=C551  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=010A   NV UP DI PL NZ NA PO CY
0CDE:010A CD20          INT     20
```

Alternatively, you can type G 10A to execute the program up to, but not including, the INT 20 instruction at 10Ah. The registers will then show the result of your program.

If you try this, you will see CX = 0 and either BX = C551 or BX = C5D1, depending on the value of the carry flag before you ran the program. The C5 your program's MOV instruction put into BL at the start is now in the BH register, but BL doesn't contain A3, because you rotated BX *through* the carry. Later, you will see other ways of rotating without going through the carry. Let's get back to the goal of printing a number in binary notation.

Building a Program to Display a Binary Number

You have seen how to strip off binary digits one at a time and convert them to ASCII characters. If you add an INT 21h instruction to print your digits, your program will be finished as you can see in the following program. The first instruction sets AH to 02 for the INT 21h function call (recall, 02 tells DOS to print the character in the DL register). The program is as follows:

```
3985:0100 B402          MOV     AH,02
3985:0102 B90800        MOV     CX,0008
3985:0105 B200          MOV     DL,00
3985:0107 D0D3          RCL     BL,1
3985:0109 80D230        ADC     DL,30
3985:010C CD21          INT     21
3985:010E E2F5          LOOP    0105
3985:0110 CD20          INT     20
```

You have seen how all the pieces work alone; here is how they work together. You rotate BL (with the instruction RCL BL,1) to pick off the bits of a number. Pick a number you want printed in binary and load it into the BL register; then run this program with a G command. After the INT 20h instruction, the G command restores the registers to the values they had before. BL still contains the number you see printed in binary.

The ADC DL,30 instruction in your program converts the carry flag to a 0 or a 1 character. The instruction MOV DL,0 sets DL to zero first, then the ADC instruction adds 30h to DL, and then finally adds the carry. Since 30h is the ASCII code for a 0, the result of ADC DL,30 is the code for 0 when the carry is clear (NC) or 1 if the carry is set (CY).

If you want to see what happens when you run this program, trace through it. Keep in mind that you will need to be a bit careful in single-stepping through it with the T command. It contains an INT 21h instruction and, as you saw when you first encountered INT 21h, DOS does a great deal of work for that one instruction. That is why you cannot use T on the INT 21.

You can, however, trace through all the other instructions in this program except the final INT 20, which won't concern you until the very end. During your tracing, type G 10E each time you loop through and reach the INT 21h instruction. Your G command, followed by an address, will tell Debug to continue running the program until IP becomes the address (10E) you entered. That is, Debug will execute the INT 21h instruction without your tracing through it, but stops before executing the LOOP instruction at 10E so you can return to tracing through the program. (The number you type after G is known as a *breakpoint*; breakpoints are very useful when you are trying to understand the inner workings of programs.) Finally, terminate the program when you reach the INT 20h instruction by typing the G command by itself.

Stepping over INTs with the Proceed Command

Whether or not you have tried out the instructions to trace through the program, you have seen that an instruction like G 10E allows you to trace *over* an INT instruction that starts at, say, 10Ch. But that means each time you want to trace over an INT instruction, you need to find the address of the instruction immediately following the INT instruction.

As it turns out, there is a Debug command that makes tracing through INT instructions much simpler. The P (*Proceed*) command does all the work for you. To see how it works, trace through the program, but when you reach the INT 21h instruction type P, rather than G 10E.

We will make heavy use of the P command in the rest of this book, because it is a nice way to trace over commands like INT, which call on large programs such as the routines inside DOS.

That is all we will cover about printing binary numbers as strings of zeros and ones for now. Here is a simple exercise for you to practice on—see if you can modify this program to print a *b* at the end of your binary number. (Hint: The ASCII code for b is 62h.)

Summary

This chapter provided a chance to catch your breath a bit after your hard work on new concepts in Chapters 1 through 3. So where have you been, and what have you seen?

You had your first encounter with flags and had a look at the carry flag. This was of special interest because it made the job of printing a binary number quite simple. It did so as soon as you learned about the rotate instruction RCL, which rotates a byte or word to the left, one bit at a time.

Once you learned about the carry flag and rotating bytes and words, you learned a new version of the add instruction, ADC. You used this later to build your program to print a number in binary notation.

This is where the LOOP instruction entered the scene. By loading the CX register with a loop count, you could keep the 80x86 executing a loop of instructions a number of times. You set CX to 8, to execute a loop eight times.

That is all you needed to write your program. You will use these tools again in the following chapters. In the next chapter we will print a binary number in hexadecimal notation, just as Debug does. By the time we finish Chapter 5, you will have a better idea of how Debug translates numbers from binary to hex. Then, we will move on to the other end of Debug—reading the numbers typed in hex and converting them to the 80x86's binary notation.

C H A P T E R

Printing Numbers in Hex

In this chapter you will learn some new tools needed to build a program that displays a binary number in hexadecimal notation. You will learn about the zero flag, as well as several other flags. You will learn how to use the CMP (compare) instruction to set these flags, and the JZ (jump if zero) conditional jump instruction to control your program. Finally, you will learn about Boolean logic and the AND instruction.

5

Topics Covered

Comparing Numbers

Printing a Single Hex Digit

Summary

The program in Chapter 4 was fairly straightforward. You were lucky because the carry flag made it easy to print a binary number as a string of 0 and 1 characters. Now we will move on to printing numbers in hex notation. The work will be a bit less direct and we will begin to repeat ourselves in our programs, writing the same sequence of instructions more than once. But that type of repetition won't last forever. In Chapter 7 you will learn about procedures or subroutines that eliminate the need to write more than one copy of a group of instructions. First, you learn some more useful instructions and see how to print numbers in hex.

Comparing Numbers

In the last chapter you learned something about status flags and examined the carry flag, which is represented as either CY or NC in Debug's R display. The other flags are equally useful, keeping track of the *status* for the last arithmetic operation. There are eight flags altogether, and you will learn about them as they are needed.

Other Status Flags

Recall that CY means the carry flag is 1, or set, whereas NC means the carry flag is 0. In all flags 1 means *true* and 0 means *false*. For example, if a SUB instruction results in 0, the flag known as the Zero Flag would be set to 1—true—and you would see it in the R display as ZR (*Zero*). Otherwise, the zero flag would be reset to 0—NZ (*Not Zero*).

Look at an example that tests the zero flag. You will use the SUB instruction to subtract two numbers. If the two numbers are equal, the result will be zero, and the zero flag will appear as ZR on your display. Enter the following subtract instruction:

```
396F:0100 29D8          SUB     AX,BX
```

Now, trace through the instruction with a few different numbers, watching for ZR or NZ to appear in the zero flag. If you place the same number (F5h in the following example) into both the AX and BX registers, you will see the zero flag set after one subtract instruction, and cleared after another, as follows:

```
-R
AX=00F5  BX=00F5  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0100   NV UP DI PL NZ NA PO NC
0CDE:0100 29D8          SUB     AX,BX
-T

AX=0000  BX=00F5  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0102   NV UP DI PL ZR NA PE NC
0CDE:0102 3F            AAS
-R IP
IP 0102
:100
-R
AX=0000  BX=00F5  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0100   NV UP DI PL ZR NA PE NC
0CDE:0100 29D8          SUB     AX,BX
-T

AX=FF0B  BX=00F5  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0102   NV UP DI NG NZ AC PO CY
0CDE:0102 3F            AAS
```

If you subtract one from zero, the result is FFFFh, which is –1 in two's-complement form. Can you tell from the R display whether a number is positive or negative? Yes—another flag, called the Sign Flag, changes between NG (*Negative*) and PL (*Plus*), and is set to 1 when a number is a negative two's-complement number.

Another new flag you will be interested in is the Overflow Flag, which changes between OV (*Overflow*) when the flag is 1 and NV (*No Overflow*) when the flag is 0. The overflow flag is set if the sign bit changes when it shouldn't. For example, if you add two positive numbers, such as 7000h and 6000h, you get a negative number, D000h, or –12288. This is an error because the result overflows the word. The result should be positive, but isn't, so the 80x86 sets the overflow flag. (Remember, if you were dealing with unsigned numbers, this would not be an error and you would ignore the overflow flag.)

Try several different numbers to see if you can set and reset each of these flags, trying them out until you are comfortable with them. For the overflow, subtract a large negative number from a large positive number—for example, 7000h – 8000h, since 8000h is a negative number equal to –32768 in two's-complement form.

Using the Status Bits—Conditional Jumps

Now you are ready to look at a set of instructions called the *conditional jump* instructions. They allow you to check status flags more conveniently than you have been able to so far. The instruction JZ (*Jump if Zero*) jumps to a new address if the last arithmetic result was zero. Thus, if you follow a SUB instruction with, say, JZ 15A, a result of zero for the subtraction would cause the 80x86 to jump to and start executing statements at address 15Ah, rather than at the next instruction.

The JZ instruction tests the zero flag, and, if it is set (ZR), does a jump. The opposite of JZ is JNZ (*Jump if Not Zero*). Let's look at a simple example that uses JNZ and subtracts one from a number until the result is zero, as follows:

```
396F:0100 2C01          SUB     AL,01
396F:0102 75FC          JNZ     0100
396F:0104 CD20          INT     20
```

Put a number like three in AL so you will go through the loop a few times, then trace through this program, one instruction at a time, to see how conditional branches work. We put the INT 20h instruction at the end so typing G by accident won't drop off the end of your program; it's a good defensive practice.

Using CMP to Compare Numbers

You may have noticed that using SUB to compare two numbers has the potentially undesirable side effect of changing the first number. CMP (*Compare*) allows you to do the subtraction without storing the result anywhere and without changing the first number. The result is used only to set the flags, so you can use one of the many conditional jump instructions after a compare. To see what happens, set both AX and BX to the same number, F5h, and trace through this instruction, as follows:

```
-A 100
0CDE:0100 CMP AX,BX
0CDE:0102
-T
```

```
AX=00F5  BX=00F5  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=0CDE  ES=0CDE  SS=0CDE  CS=0CDE  IP=0102   NV UP DI PL ZR NA PE NC
0CDE:0102 3F            AAS
```

The zero flag is now set (ZR), but F5h remains in both registers.

To use CMP to print a single hex digit. Create a set of instructions that uses flags to alter the flow of your program, as LOOP did in the last chapter, in a manner similar to BASIC's IF-THEN statement. This new set of instructions will use the flags to test for such conditions as less than, greater than, and so on. You will not have to worry about which flags are set when the first number is less than the second; the instructions know which flags to look at.

Printing a Single Hex Digit

Start by putting a small number (between 0 and Fh) into the BL register. Since any number between 0 and Fh is equivalent to one hex digit, you can convert your choice to a single ASCII character and then print it. Look at the steps needed to do the conversion.

Character	ASCII Code (Hex)	Character	ASCII Code (Hex)
/	2F	<	3C
0	30	=	3D
1	31	>	3E
2	32	?	3F
3	33	@	40
4	34	A	41
5	35	B	42
6	36	C	43
7	37	D	44
8	38	E	45
9	39	F	46
:	3A	G	47
;	3B	H	49

Figure 5-1: Partial ASCII table showing the characters used by hex digits.

The ASCII characters 0 through 9 have the values 30h through 39h; the characters A through F, however, have the values 41h through 46h (see Figure 5-1).

The problem is that these two groups of ASCII characters are separated by seven characters. As a result, the conversion to ASCII will be different for the two groups of numbers (0 through 9 and Ah through Fh), so you handle each group differently. A BASIC program to do this two-part conversion looks like this:

```
IF  BL < &H0A THEN
    BL = BL + &H30
ELSE
    BL = BL + &H37
END IF
```

The BASIC conversion program is fairly simple. Unfortunately, the 80x86's machine language doesn't include an ELSE statement. It is far more primitive than BASIC is, so you will need to be somewhat clever. Here is another BASIC program, this time one that mimics the method you will use for the machine-language program.

```
BL = BL + &H30
IF  BL >= &H3A THEN
    BL = BL + &H7
END IF
```

You can convince yourself that this program works by trying it with some choice examples. The numbers 0, 9, Ah, and Fh are particularly good because these four numbers cover all the *boundary* conditions, which are areas where we often run into problems.

Notice that we just wrote 0Ah for the number A, rather than AH, so we wouldn't confuse the number Ah with the register AH. We will often place a zero before hex numbers in situations like this that could be confusing. In fact, since it never hurts to place a zero before a hex number, it is a good idea to place a zero before all hex numbers.

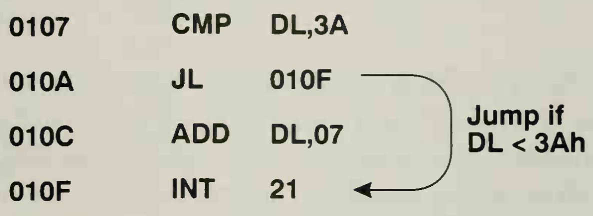
Here, 0 and Fh are, respectively, the smallest and largest single-digit hex numbers. By using 0 and Fh, you check the bottom and top of the range. The numbers 9 and 0Ah, although next to each other, require two different conversion schemes in the program. By using 9 and 0Ah, you confirm that you have chosen the correct place to switch between these two conversion schemes.

The machine-language version of this program contains a few more steps, but it is essentially the same as the BASIC version. It uses the CMP instruction, as well as a conditional jump instruction called JL (*Jump if Less Than*). The program to take a single-digit hex number in the BL register and print it in hex is as follows:

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 80C230        ADD     DL,30
3985:0107 80FA3A        CMP     DL,3A
```

```
3985:010A 7C03          JL      010F
3985:010C 80C207        ADD     DL,07
3985:010F CD21          INT     21
3985:0111 CD20          INT     20
```

The CMP instruction, as you saw before, subtracts two numbers (DL – 3Ah) to set the flags, but it does not change DL. So if DL is less than 3Ah, the JL 10F instruction skips to the INT 21h instruction at 10Fh. Place a single-digit hex number in BL and trace through this example to get a better feeling for CMP and our algorithm to convert hex to ASCII. Remember to use either the G command with a breakpoint or the P command when you run the INT instructions.



```
0107      CMP   DL,3A
010A      JL    010F
010C      ADD   DL,07
010F      INT   21
```

Figure 5-2: The JL instruction.

Using Rotate To Get the Upper Nibble

The program works for any single-digit hex number, but we need a few more steps to print a two-digit hex number. We need to isolate each digit (four bits, which are often called a *nibble*) of this two-digit hex number. In this section, you will see that you can easily isolate the first, or higher, four bits. In the next section, you will encounter a concept known as a *logical operation*, which you will use to isolate the lower four bits—the second of the two hex digits.

To begin, recall that the RCL instruction rotates a byte or a word to the left through the carry flag. In the last chapter you used the instruction RCL BL,1, where the number one told the 80x86 to rotate BL left by one bit. You can rotate by more than one bit if you want, but you cannot simply write the instruction RCL BL,2. For rotations by more than one bit, you must place a rotate count in the CL register.

Note

> Although RCL BL,2 isn't a legal 8088 instruction, it works just fine with the 80286 and above processors found on most of today's computers. Since there are still some PCs with the older 8088s, it is best to write your programs for the lowest common denominator—the older 8088.

The CL register is used here in much the same way as the CX register is used by the LOOP instruction to determine the number of times to repeat a loop. The 80x86 uses CL for the number of times to rotate a byte or word, rather than the CX register, because it makes no sense to rotate more than 16 times. The eight-bit CL register is more than large enough to hold your maximum shift count.

How does all this tie in with printing a two-digit hex number? The plan is to rotate the byte in DL four bits to the right. To do so, you will use a slightly different rotate instruction called SHR (*Shift Right*). Using SHR, you will be able to move the upper four bits of your number to the rightmost nibble (four bits).

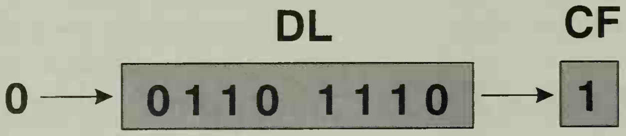


Figure 5-3: The SHR DL,1 Instruction.

You also want the upper four bits of DL set to zero, so that the entire register becomes equal to the nibble you are shifting into the right nibble. If you were to enter SHR DL,1, your instruction would move the byte in DL one bit to the right. At the *same* time it would move bit 0 into the carry flag, while *shifting* a zero into bit 7 (the highest, or leftmost, bit in DL). If you do that three more times, the upper four bits will end up shifted right into the lower four bits, while the upper four bits will all have had zeros shifted into them. You can do all that shifting in one instruction by using the CL register as the *shift count*. Setting CL to four before the instruction SHR DL,CL, will ensure that DL becomes equal to the upper hex digit. To see how this works, place 4 into

CL and 5Dh into DL. Then enter and trace through the following SHR instruction:

```
3985:0100 D2EA          SHR     DL,CL
```

DL should now be 05h, which is the first digit in the number 5Dh. Now you can print this digit with a program like the one used earlier. By putting together the pieces you have so far, you can build the following program to take a number in the BL register and print the first hex digit:

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 B104          MOV     CL,04
3985:0106 D2EA          SHR     DL,CL
3985:0108 80C230        ADD     DL,30
3985:010B 80FA3A        CMP     DL,3A
3985:010E 7C03          JL      0113
3985:0110 80C207        ADD     DL,07
3985:0113 CD21          INT     21
3985:0115 CD20          INT     20
```

Using AND to Isolate the Lower Nibble

Now that you can print the first of the two digits in a hex number, let's see how you can isolate and print the second digit. You will clear the upper four bits of your original (unshifted) number to zero, leaving DL equal to the lower four bits. It is easy to set the upper four bits to zero with an instruction called AND. The AND instruction is one of the *logical* instructions—those that have their roots in formal logic.

In formal logic, "A is true, if B *and* C are both true." But if either B or C is false, then A must also be false. If you take this statement, substitute one for true and zero for false, then look at the various combinations of A, B, and C, you can create what is known as a *truth table*. Here is the truth table for ANDing two bits together:

AND	F	T
F	F	T
T	F	T

=

AND	0	1
0	0	0
1	0	1

Down the left and across the top are the values for the two bits. The results for the AND are in the table, so you see that 0 AND 1 gives 0.

The AND instruction works on bytes and words by ANDing together the bits of each byte or word that are in the same position. For example, the statement AND BL,CL successively ANDs bits 0 of BL and CL, bits 1, bits 2, and so on, and places the result in BL. We can make this clearer with an example in binary, as follows:

```
     1 0 1 1 0 1 0 1
AND  0 1 1 1 0 1 1 0
--------------------
     0 0 1 1 0 1 0 0
```

Furthermore, by ANDing 0Fh to any number, you can set the upper four bits to zero, as follows:

```
     0 1 1 1 1 0 1 1     7Bh
AND  0 0 0 0 1 1 1 1  =  0Fh
----------------------------
     0 0 0 0 1 0 1 1     0Bh
```

Next you can put this logic into a short program that takes the number in BL, isolates the lower hex digit by ANDing 0Fh to the upper four bits, and then prints the result as a character. You saw most of the details of this program when you printed the upper hex digit; the only new detail is the AND instruction.

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 80E20F        AND     DL,0F
3985:0107 80C230        ADD     DL,30
3985:010A 80FA3A        CMP     DL,3A
3985:010D 7C03          JL      0112
3985:010F 80C207        ADD     DL,07
3985:0112 CD21          INT     21
3985:0114 CD20          INT     20
```

Try this with some two-digit hex numbers in BL before you move on to put the pieces together to print both digits. You should see the rightmost hex digit of your number in BL on the screen.

Putting It All Together

There isn't much to change when you put all the pieces together. You need only to change the address of the second JL instruction used to print the second hex digit. The complete program is as follows:

```
3985:0100 B402          MOV     AH,02
3985:0102 88DA          MOV     DL,BL
3985:0104 B104          MOV     CL,04
3985:0106 D2EA          SHR     DL,CL
3985:0108 80C230        ADD     DL,30
3985:010B 80FA3A        CMP     DL,3A
3985:010E 7C03          JL      0113
3985:0110 80C207        ADD     DL,07
3985:0113 CD21          INT     21
3985:0115 88DA          MOV     DL,BL
3985:0117 80E20F        AND     DL,0F
3985:011A 80C230        ADD     DL,30
3985:011D 80FA3A        CMP     DL,3A
3985:0120 7C03          JL      0125
3985:0122 80C207        ADD     DL,07
3985:0125 CD21          INT     21
3985:0127 CD20          INT     20
```

After you have entered this program, you will have to type *U 100*, followed by *U*, to see the entire unassembled listing. Note that you have repeated one set of five instructions twice: the instructions at 108h through 113h, and 11Ah through 125h. In Chapter 7 you will learn how to write this sequence of instructions just once using a new instruction called CALL.

Summary

In this chapter, you learned more about how Debug translates numbers from the 80x86's binary format to a hex format you can read. What did you add to your growing store of knowledge?

First, you learned about some of the two-letter flags you see on the right side of the register (R) display. These status bits give you a great deal of information about the last arithmetic operation. By looking at the zero flag, for example, you could tell whether the result of the last operation was zero. You also found that you could compare two numbers with a CMP instruction.

Next, you learned how to print a single-digit hex number. Armed with this information, you learned about the SHR instruction, enabling you to move the upper digit of a two-digit hex number into the lower four bits of BL. That completed, you could print the digit, as before.

Finally, the AND instruction allowed you to isolate the lower hex digit from the upper. By putting all these pieces together, you wrote a program to print a two-digit hex number.

We could have continued on to print a four-digit hex number, but at this point, we would find ourselves repeating instructions. Before you try to print a four-digit hex number, you will learn about procedures in Chapter 7. Then, you will know enough to write a procedure to do the job. By then you will also be ready to learn about the assembler—a program that will do much of your work. Now, on to reading hex numbers.

C H A P T E R

Reading Characters

In this chapter you will learn how to read characters from your keyboard. You will build a small program that reads a two-digit hex number from the keyboard.

6

Topics Covered

Now that you know how to print a byte in hex notation, you are going to reverse the process by reading two characters—hex digits—from the keyboard and converting them into a single byte.

Reading One Character

The DOS INT 21h function call we have been using has an input function, number 1, that reads a character from the keyboard. When you learned about function calls in Chapter 4, you saw that the function number must be placed in the AH register before an INT 21h call. To try function 1 for INT 21h, enter INT 21h at location 0100h as follows:

```
396F:0100 CD21          INT     21
```

Place 01h into AH and type either *G 102* or *P* to run this one instruction. All you will see is the blinking cursor. Actually, DOS has paused and is waiting until you press a key (don't do so yet). Once you press a key, DOS will place the ASCII code for that character into the AL register. You will use this instruction later to read the characters of a hex number. Right now, see what happens when you press a key like the F1 key. DOS will return a 0 in AL and a semicolon (;) after Debug's hyphen prompt.

F1 is one of a set of special keys with *extended codes*, which DOS treats differently from the keys representing normal ASCII characters. (You will find a table listing these extended codes in Appendix D.) For each of these special keys, DOS sends *two* characters, one right after the other. The first character returned is always zero, indicating that the next character is the *scan code* for a special key.

To read both characters, you would have to execute INT 21h twice. But in our example, you read only the first character, the zero, and leave the scan code in DOS. When Debug finished with the G 102 (or P) command, it began to read characters, and the first character it read was the scan code left behind from the F1 key, namely, 59, which is the ASCII code for a semicolon.

Later, when we develop the Dskpatch program, you will begin to use these extended codes to bring the cursor and function keys to life. Until then, we will just work with the normal ASCII characters.

Reading a Single-Digit Hex Number

Next you need to reverse the conversion that you used in Chapter 5, when you transformed a single-digit hex number to the ASCII code for one of the characters in 0 through 9 or A through F. To convert one character, such as C or D, from a hex character to a byte, you must subtract either 30h (for 0 through 9) or 37h (for A through F). Here is a simple program that will read one ASCII character and convert it to a byte.

```
3985:0100 B401          MOV     AH,01
3985:0102 CD21          INT     21
3985:0104 2C30          SUB     AL,30
3985:0106 3C09          CMP     AL,09
3985:0108 7E02          JLE     010C
3985:010A 2C07          SUB     AL,07
3985:010C CD20          INT     20
```

Most of these instructions should be familiar now, but there is one new one, JLE (*Jump if Less than or Equal*). In your program, this instruction jumps if AL is less than or equal to 9h.

To see the conversion from hex character to ASCII, you need to see the AL register just before the INT 20h is executed. Since Debug restores the registers when it executes the INT 20h, you will need to set a breakpoint, as you did in Chapter 4. Here, type *G 10C*, and you will see that AL will contain the hex number converted from a character.

Try typing some characters, such as *k* or a lowercase *d*, that are not hex digits to see what happens. You will notice that this program works correctly only when the input is one of the digits 0 through 9 or the uppercase letters A through F. You will correct this minor failing in the next chapter when you learn about subroutines, or procedures. Until then, we will be sloppy and ignore error conditions. You will have to type correct characters for your program to work properly.

Reading a Two-Digit Hex Number

Reading two hex digits isn't much more complicated than reading one, but it does require many more instructions. Begin by reading the first digit, then place its hex value in the DL register and multiply it by 16. To perform this multiplication, shift the DL register left four bits, which places a hex zero (four zero bits) to the right of the digit you just read. The instruction SHL DL,CL, with CL set to four, does the trick by inserting zeros at the right. In fact, the SHL (*Shift Left*) instruction is known as an *arithmetic shift*, because it has the same effect as an arithmetic multiplication by two, four, eight, and so on, depending on the number (such as one, two, or three) in CL (See Figure 6-1).

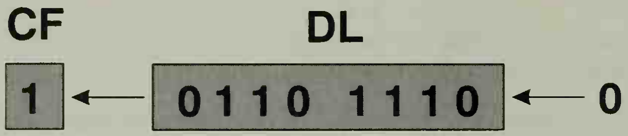


Figure 6-1: The SHL DL,1 instruction.

Finally, with the first digit shifted over, add the second hex digit to the number in DL (the first digit * 16). You can see and work through all these details in the following program:

```
3985:0100 B401          MOV     AH,01
3985:0102 CD21          INT     21
3985:0104 88C2          MOV     DL,AL
3985:0106 80EA30        SUB     DL,30
3985:0109 80FA09        CMP     DL,09
3985:010C 7E03          JLE     0111
3985:010E 80EA07        SUB     DL,07
3985:0111 B104          MOV     CL,04
3985:0113 D2E2          SHL     DL,CL
3985:0115 CD21          INT     21
3985:0117 2C30          SUB     AL,30
3985:0119 3C09          CMP     AL,09
3985:011B 7E02          JLE     011F
3985:011D 2C07          SUB     AL,07
3985:011F 00C2          ADD     DL,AL
3985:0121 CD20          INT     20
```

Now that you have a working program, it is a good idea to check the boundary conditions to confirm that it is working properly. For these boundary conditions, use the numbers 00, 09, 0A, 0F, 90, A0, F0, and some other number, such as 3C.

Use a breakpoint to run the program without executing the INT 20h instruction. (Make sure you use uppercase letters for your hex input.)

Summary

You have finally had a chance to practice what you learned in previous chapters without being flooded with new information. Using a new INT 21 function (number 1) to read characters, you developed a program to read a two-digit hex number. Along the way, we emphasized the need to test programs with all the boundary conditions.

We will wrap up Part I by learning about procedures in the 80x86.

CHAPTER

Using Procedures to Write Reusable Code

In this chapter you will learn how to write small, general-purpose pieces of code called procedures that you can use from anywhere in your program. Procedures are something you will use heavily in almost any program you write. In this process you will also learn about the stack, and you will build a new hex-input routine that prevents you from typing invalid digits.

7

Topics Covered

In the next chapter, we will discuss MASM, the macro assembler, and you will begin to use *assembly* language. But before leaving Debug, you will look at one last set of examples and learn about subroutines and a special place to store numbers called the *stack.*

Writing Procedures

A procedure is a list of instructions that can be executed from different places in a program, rather than having to repeat the same list of instructions at each place they're needed. In BASIC such lists are called *subroutines,* but we will call them *procedures* for reasons that will become clear later.

You move to and from procedures by using a pair of instructions. You call a procedure with one instruction, *CALL.* You return from the procedure with a *RET* instruction.

Here is a simple BASIC program, written in QBasic, that you will later rewrite in machine language. This program calls a subroutine ten times, each time printing a single character, starting with A and ending with J, as follows:

```
DIM SHARED A
A = &H41                    'ASCII for 'A'
FOR I = 1 TO 10
    DoPrint
    A = A + 1
NEXT I

SUB DoPrint
    PRINT CHR$(A);
END SUB
```

QBasic is actually much simpler than older versions of BASIC, such as GW-BASIC, that were shipped with versions of DOS before DOS 5.0. Earlier versions of BASIC required line numbers in front of each line of the program, and you had to use the GOSUB and RETURN commands to write the same code. Since the older BASIC looks more like the machine code we will write below, here is the above program rewritten in a primitive version of BASIC, using line numbers.

```
10 A = &H41                    'ASCII for 'A'
20 FOR I = 1 TO 10
30 GOSUB 1000
40 A = A + 1
50 NEXT I
60 END

1000 PRINT CHR$(A)
1010 RETURN
```

In this case the subroutine follows a practice common when BASIC programs needed line numbers, by beginning at line 1000 to leave room for more instructions to be added to the main program without affecting the line number of the subroutine. You will have to do the same with your machine-language procedure since you cannot move machine-language programs easily (this won't be a problem once you start to use the assembler in the next chapter, so it is only a temporary problem). You will put the machine-language subroutine at 200h, far away from the main program at 100h. You will also replace GOSUB 1000 with the instruction CALL 200h, which *calls* the procedure at memory location 200h. The CALL sets IP to 200h, and the 80x86 starts executing the instructions at 200h.

The FOR-NEXT loop of the BASIC program (as you saw in Chapter 4) can be written as a LOOP instruction. The other pieces of the main program (except for the INC instruction) should be familiar. They are as follows:

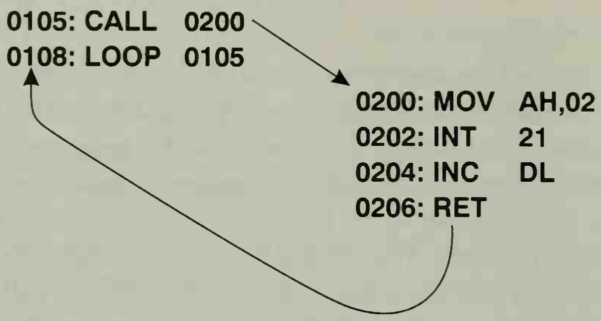
```
3985:0100 B241          MOV     DL,41
3985:0102 B90A00        MOV     CX,000A
3985:0105 E8F800        CALL    0200
3985:0108 FEC2          INC     DL
3985:010A E2F9          LOOP    0105
3985:010C CD20          INT     20
```

The first instruction places 41h (ASCII for A) into the DL register, because the INT 21h instruction prints the character given by the ASCII code in DL. The INT 21h instruction itself is located some distance away, in the procedure at location 200h. INC DL, the new instruction, *increments* the DL register. That is, it adds one to DL, setting DL to the next character in the alphabet. The procedure you should enter at 200h is as follows:

```
3985:0200 B402          MOV     AH,02
3985:0202 CD21          INT     21
3985:0204 C3            RET
```

Recall that the 02h in AH tells DOS to print the character in DL when you execute the INT 21h instruction. RET is a new instruction that *returns* to the first instruction (LOOP) following the CALL in your main program.

Type G to see the output of this program, then single-step through it to see how it works (remember to use either a breakpoint or the P command to run the INT 21 instruction).



```
0105: CALL  0200
0108: LOOP  0105
                       0200: MOV   AH,02
                       0202: INT   21
                       0204: INC   DL
                       0206: RET
```

Figure 7-1: The CALL and RET instructions.

How CALL Works: The Stack and Return Addresses

The CALL instruction in your program needs to save the *return address* somewhere so the 80x86 will know where to resume executing instructions when it sees the RET instruction. For the storage place itself, there is a portion of memory known as the *stack.* For tracking what is on the stack, there are two registers that you can see on the R display: the SP (*Stack Pointer*) register, which points to the top of the stack, and the SS (*Stack Segment*), which holds the segment number.

In operation, a stack for the 80x86 is just like a stack of trays in a cafeteria, where placing a tray on the top covers the trays underneath. The last tray on the stack is the first to come off, so another name for a stack is LIFO, for *Last*

In, First Out. This order, LIFO, is precisely what you need for retrieving return addresses after you make *nested* CALLs like the following:

```
396F:0100 E8FD00        CALL    0200
         .
         .
         .
396F:0200 E8FD00        CALL    0300
396F:0203 C3            RET
         .
         .
         .
396F:0300 E8FD00        CALL    0400
396F:0303 C3            RET
         .
         .
         .
396F:0400 C3            RET
```

The instruction at 100h calls one at 200h, which calls one at 300h, which calls one at 400h, where you finally see a return (RET) instruction. This RET returns to the instruction following the *previous* CALL instruction, at 300h, so the 80x86 resumes executing instructions at 303h. But there it encounters a RET instruction at 303h, which pulls the next oldest address (203h) off the stack. So the 80x86 resumes executing instructions at 203h, and so on. Each RET *pops* the topmost return address off the stack, so each RET follows the same path backward as the CALLs did forward, see Figures 7-2 and 7-3.

Try entering a program like the preceding one. Use multiple calls, and trace through the program to see how the calls and returns work. Although the process may not seem very interesting right now, there are other uses for this stack, and a good understanding of how it works will come in handy. (In a later chapter, you will look for the stack in memory.)

PUSHing and POPping Data

The stack is a useful place to store words of data for a while, provided you are careful to restore the stack before a RET instruction. You have seen that a CALL instruction *pushes* the return address (one word) onto the top of the stack, while a RET instruction *pops* this word off the top of the stack, loads it into the IP register, and exposes the word that was lying underneath it. You can do the same thing with the instructions PUSH and POP, which allow you to push and pop words.

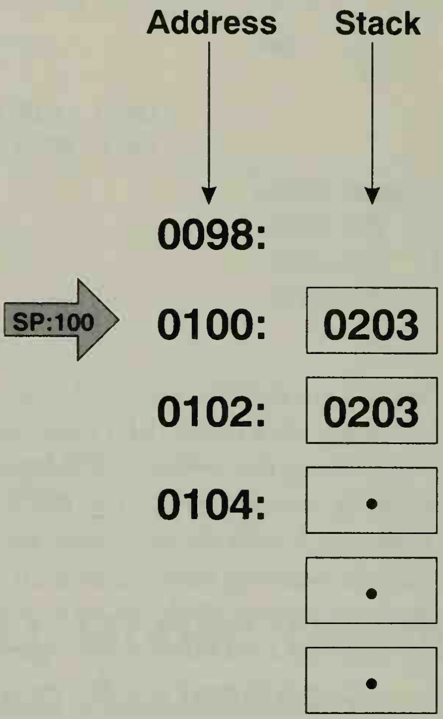


Figure 7-2: The stack just before executing the CALL 400 instruction.

It is often convenient to save the values of registers at the beginning of a procedure and restore them at the end, just before the RET instruction. Then you are free to use these registers in any way you like within the procedure, as long as you restore their values at the end.

Programs are built from many levels of procedures, with each level calling the procedures at the next level down. By saving registers at the beginning of a procedure and restoring them at the end, you remove unwanted interactions between procedures at different levels, making your programming much easier. You will see more about saving and restoring registers in Chapter 13, when we discuss modular design. Following is an example (don't enter it) that saves and restores CX and DX:

```
396F:0200 51            PUSH   CX
396F:0201 52            PUSH   DX
396F:0202 B90800        MOV    CX,0008
396F:0205 E8F800        CALL   0300
396F:0208 FEC2          INC    DL
396F:020A E2F9          LOOP   0205
396F:020C 5A            POP    DX
396F:020D 59            POP    CX
396F:020E C3            RET
```

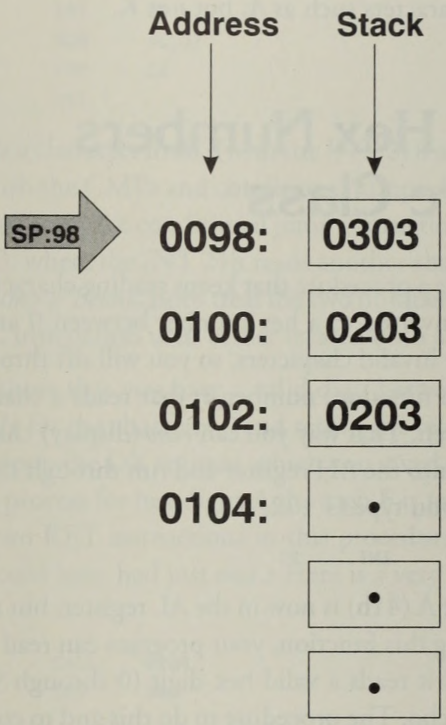


Figure 7-3: The stack just after executing the CALL 400 instruction.

Notice that the POPs are in reverse order from the PUSHes, because a POP removes the word placed most recently on the stack, and the old value of DX is on top of the old CX.

Saving and restoring CX and DX allows you to change these registers in the procedure that begins at 200h, but without changing the values used by any procedure that calls this one. Once you have saved CX and DX, you can use these registers to hold *local* variables—variables you can use within this procedure without affecting the values used by the calling program.

You will often use such local variables to simplify programming tasks. As long as you are careful to restore the original values, you won't have to worry about your procedures changing any of the registers used by the calling program. This will become clearer in the next example, which is a procedure to read a hex number. Unlike the program in Chapter 6, your new program now will allow only valid characters such as A, but not K.

Reading Hex Numbers with More Class

Next, you can create a procedure that keeps reading characters until it receives one that can be converted to a hex number between 0 and Fh. You do not want to display any invalid characters, so you will sift through your input using a new INT 21h function, number 8, that reads a character but does not pass it onto the screen. That way you can *echo* (display) characters only if they are valid. Place 8h into the AH register and run through this instruction, typing an A just after you type G 102.

```
3985:0100 CD21          INT     21
```

The ASCII code for A (41h) is now in the AL register, but the A didn't appear on the screen. Using this function, your program can read characters without echoing them until it reads a valid hex digit (0 through 9 or A through F), which it will then echo. The procedure to do this and to convert the hex character to a hex number is as follows:

```
3985:0200 52            PUSH    DX
3985:0201 B408          MOV     AH,08
3985:0203 CD21          INT     21
3985:0205 3C30          CMP     AL,30
3985:0207 72FA          JB      0203
3985:0209 3C46          CMP     AL,46
```

```
3985:020B 77F6          JA      0203
3985:020D 3C39          CMP     AL,39
3985:020F 770A          JA      021B
3985:0211 B402          MOV     AH,02
3985:0213 88C2          MOV     DL,AL
3985:0215 CD21          INT     21
3985:0217 2C30          SUB     AL,30
3985:0219 5A            POP     DX
3985:021A C3            RET
3985:021B 3C41          CMP     AL,41
3985:021D 72E4          JB      0203
3985:021F B402          MOV     AH,02
3985:0221 88C2          MOV     DL,AL
3985:0223 CD21          INT     21
3985:0225 2C37          SUB     AL,37
3985:0227 5A            POP     DX
3985:0228 C3            RET
```

The procedure reads a character in AL (with the INT 21h at 203h) and checks to see if it is valid with the CMPs and conditional jumps. If the character just read is not a valid character, the conditional jump instructions sends the 80x86 back to location 203, where the INT 21h reads another character. (JA is *Jump if Above*, and JB is *Jump if Below*. Both treat the two numbers as unsigned numbers, whereas the JL instruction used earlier treated both as signed numbers.)

By line 211h, you know that you have a valid digit between 0 and 9, so you just subtract the code for the character 0 and return the result in the AL register, remembering to pop the DX register, which you saved at the beginning of the procedure. The process for hex digits A through F is much the same. Notice that there are two RET instructions in this procedure. (You could have had more, or you could have had just one.) Here is a very simple program to test the procedure:

```
3985:0100 E8FD00        CALL    0200
3985:0103 CD20          INT     20
```

As you have done before, use either the G command with a breakpoint, or use the P command. You want to execute the CALL 200h instruction without executing the INT 20h instruction, so you can see the registers just before the program terminates and the registers are restored.

You will see the cursor at the left side of the screen, waiting patiently for a character. Type *k*, which isn't a valid character. Nothing should happen. Now, type any of the uppercase hex characters. You should see the character's hex value in AL and the character itself echoed on the screen. Test this procedure with the boundary conditions: '/' (the character before zero), 0, 9, ':' (the character just after 9), and so on. Now that you have this procedure, the program to read a two-digit hex number with error handling is fairly straightforward.

```
3985:0100 E8FD00        CALL    0200
3985:0103 88C2          MOV     DL,AL
3985:0105 B104          MOV     CL,04
3985:0107 D2E2          SHL     DL,CL
3985:0109 E8F400        CALL    0200
3985:010C 00C2          ADD     DL,AL
3985:010E B402          MOV     AH,02
3985:0110 CD21          INT     21
3985:0112 CD20          INT     20
```

You can run this program from DOS, since it reads in a two-digit hex number and then displays the ASCII character that corresponds to the number you typed in.

Aside from the procedure, your main program is much simpler than the version you wrote in the last chapter, and you have not duplicated the instructions to read characters. You did add error handling, even if it did complicate the procedure; error handling ensures that the program accepts only valid input.

You can also see the reason for saving the DX register in the procedure. The main program stores the hex number in DL, so you don't want your procedure at 200h to change DL. On the other hand, the procedure at 200h must use DL itself to echo characters. So, by using the instruction PUSH DX at the beginning of the procedure, and POP DX at the end, you saved yourself from problems. From now on, to avoid complicated interactions between procedures, be very strict about saving any registers used by a procedure.

Summary

Your programming is becoming more sophisticated. You have learned about procedures, which enable you to reuse the same set of instructions without rewriting them each time. You have also discovered the stack and seen that a CALL stores a return address on the top of the stack, while a RET instruction returns to the address on the top of the stack.

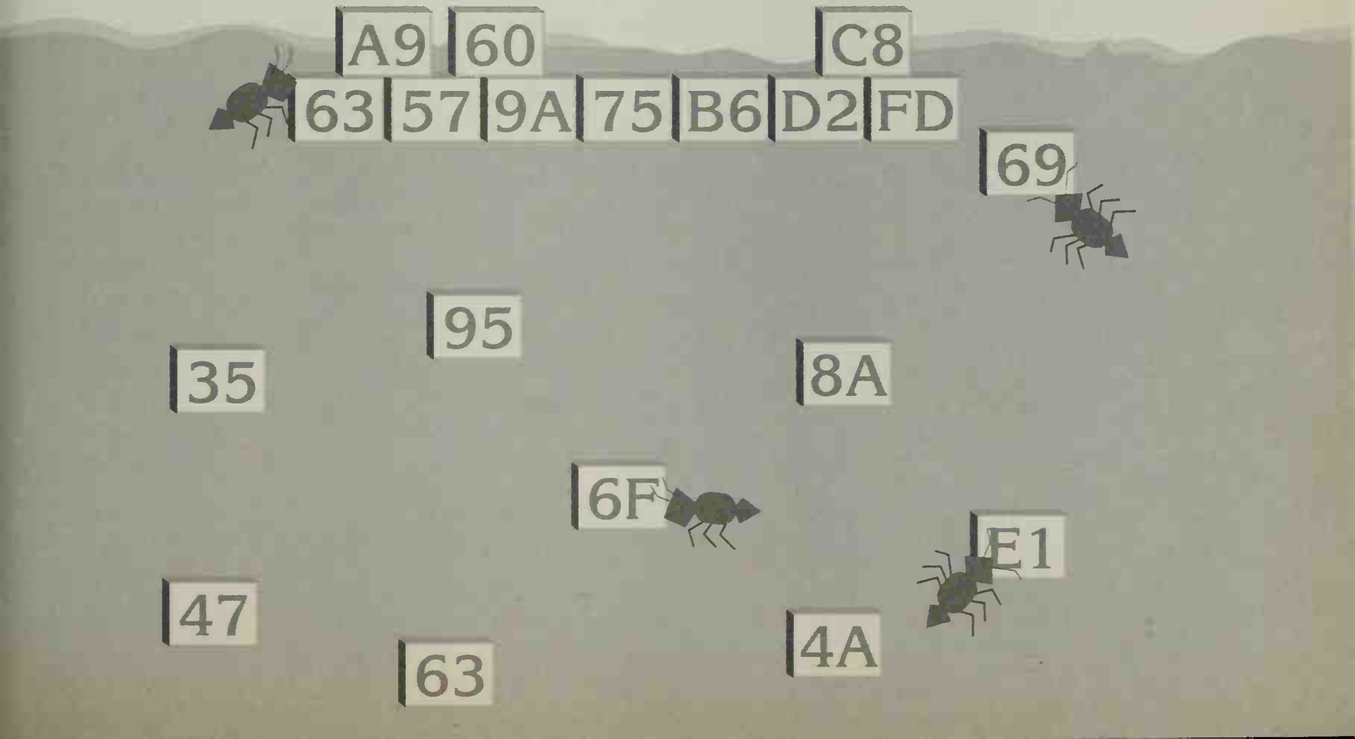
You learned how to use the stack for more than just saving return addresses. You used the stack to store the values of registers (with a PUSH instruction) so you could use them in a procedure. By restoring the registers (with a POP instruction) at the end of each procedure, you avoided unwanted interactions between procedures. By always saving and restoring registers in procedures that you write, you can CALL other procedures without worrying about which registers are used within the other procedure.

Finally, armed with this knowledge, you moved on to build a better program to read hex numbers—this time, with error checking. The program you built here is similar to one you will use in later chapters, when you begin to develop your Dskpatch program.

Now you are ready to move on to Part II, where you will learn how to use the assembler. The next chapter covers the use of the assembler to convert a program to machine language. You will also see that there will not be any reason to leave room between procedures, as in this chapter, when you put your procedure way up at location 200h.

PART II

Assembly Language



CHAPTER

Welcome to the Assembler

In this chapter you will learn how to build programs using either the Microsoft Macro Assembler or the Borland Turbo Assembler. Building programs with the assembler is much easier than building them with Debug. You will learn how to assemble and link programs and how to comment your programs and use labels so they are easy to understand.

8

Topics Covered

Building a Program without Debug

Creating Source Files

Linking Your Program

Looking at Writestr in Debug

Using Comments

Using Labels for Code

Summary

At long last you are ready to meet the assembler, a DOS program that will make your programming much simpler. From now on, you will write mnemonic, human-readable instructions directly, using the assembler to turn your programs into machine code.

This chapter and the next will be somewhat heavy with details on the assembler, but learning these details will be well worth the effort. Once you know how to use the assembler, you will get back on course in learning how to write assembly-language programs.

Building a Program without Debug

Up to this point, you have typed only *DEBUG* and then your program instructions. Now that you are about to leave Debug behind and to write programs without it, you will have to use either an editor or a word processor to create text, or human-readable files containing assembly-language instructions.

Begin by creating a *source file*—the name for the text version of an assembly-language program. You will create a source file for the program you built and named Writestr (short for Write Star) back in Chapter 3. To refresh your memory, here is the Debug version:

```
396F:0100 B402          MOV     AH,02
396F:0102 B261          MOV     DL,2A
396F:0104 CD21          INT     21
396F:0106 CD20          INT     20
```

Use a text editor, such as DOS's Edit, to enter the following lines of code into a file named WRITESTR.ASM (the extension .ASM means this is an assembler source file). Here, as with Debug, lowercase works just as well as uppercase, but we will continue to use uppercase letters to avoid confusion between the number 1 (one) and the lowercase letter l:

```
.MODEL SMALL
.CODE

        MOV     AH,2h
        MOV     DL,2Ah
        INT     21h
        INT     20h

        END
```

This is the same program you created in Chapter 3, but it contains a few necessary changes and additions. Notice that there is an *h* after each hex number in the program. This *h* tells the assembler that the numbers are in hexadecimal. Unlike Debug, which assumes all numbers are in hexadecimal, the assembler assumes all numbers are decimal. You tell it otherwise by placing an *h* after any hexadecimal number (see Figure 8-1).

Warning

The assembler can become confused by numbers, such as ACh, that look like a name or an instruction. To avoid this, always type a zero before a hex number that begins with a letter. For example, type 0ACh—*not* ACh.

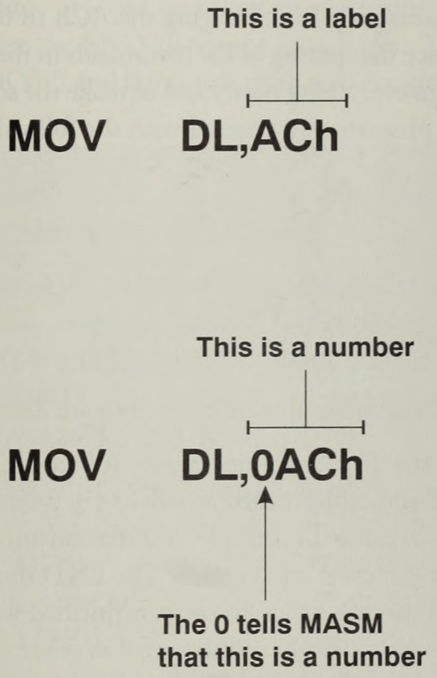


Figure 8-1: Put a zero before hexadecimal numbers starting with a letter, otherwise the assembler will treat the number as a name.

The following program demonstrates what happens when you assemble a program with ACH rather than 0ACH.

```
.MODEL  SMALL
.CODE

        MOV     DL,ACh
        INT     20h

        END
```

The output is as follows:

```
C>ML /C TEST.ASM
Microsoft (R) Macro Assembler Version 6.00
Copyright (C) Microsoft Corp 1981-1991.  All rights reserved.

 Assembling: test.asm
test.asm(3): error A2006: undefined symbol : ACh

C>
```

Definitely not encouraging. But changing the ACh to 0ACh will satisfy the assembler. Also notice the spacing of the commands in the assembler program. We used tabs to align everything neatly and to make the source text more readable. Compare the program you entered with the following:

```
.MODEL SMALL
.CODE
MOV AH,2h
MOV DL,2Ah
INT 21h
INT 20h
END
```

A bit of a mess; the assembler does not care, but we do.

Now let's return to the three new lines in your source file. The three new lines are all *directives* (also sometimes called *pseudo-ops*, or pseudo-operations). They are called directives because they supply information and directions to the assembler, rather than generate instructions. The END directive marks the end of the source file so the assembler knows it is finished when it sees an END. Later on, you will see that END is also useful in other ways. But right now, let's put aside any further discussion of it or the other two directives and focus on the assembler.

Creating Source Files

Even though you have entered the lines of WRITESTR.ASM, there is one more consideration before we move on to actually assemble our program. The assembler can use source files that contain standard ASCII characters only. If you are using a word processor, bear in mind that not all word processors write disk files using only the standard ASCII characters. Microsoft Word is one such culprit. For such word processors, use the non-document, or unformatted, mode when you save your files.

Before you try assembling WRITESTR.ASM, make sure it is still ASCII. From DOS, type:

```
C>TYPE WRITESTR.ASM
```

You should see the same text you entered, as you entered it. If you see strange characters in your program (many word processors put additional formatting information into the file, which the assembler will treat as errors) you may have to use a different editor or word processor to enter programs. You will also need a blank line after the END statement in your file. The DOS Edit program (available in DOS 5 and later) will work well for creating source files.

Note

You will need to read this note if you are using a version of MASM before 6.0, or Borland's Turbo Assembler.

If you are using an older version of MASM than 6.0, you will need to use a slightly different syntax for assembling files. Instead of using a command like "ML /C WRITESTR.ASM", you will need to use the command MASM followed by the file name (without the .ASM extension), and finally a semicolon: "MASM WRITESTR;".

If you are using Borland's Turbo Assembler, you will need to use a slightly different syntax for assembling files. Instead of using a command like "ML /C WRITESTR.ASM", you will need to use the command TASM followed by the file name (without the .ASM extension), and finally a semicolon: "TASM WRITESTR;".

Now, let's begin to assemble Writestr. (If you are using Borland's Turbo Assembler, type "TASM WRITESTR;" instead of "ML /C WRITESTR.ASM".)

```
C>ML /C WRITESTR.ASM
Microsoft (R) Macro Assembler Version 6.00
Copyright (C) Microsoft Corp 1981-1991.  All rights reserved.

 Assembling: writestr.asm

C>
```

You are not finished yet. At this point, the assembler has produced a file called WRITESTR.OBJ, which you will find on your disk. This is an intermediate file, called an *object file.* It contains the machine-language program, along with a lot of bookkeeping information used by another DOS program called the *Linker.*

Linking Your Program

Right now, you want the linker to take your OBJ file and create an EXE version of it. Link WRITESTR.OBJ by typing the following:

```
C>LINK WRITESTR;

Microsoft (R) Segmented-Executable Linker  Version 5.13
Copyright (C) Microsoft Corp 1984-1991.  All rights reserved.

LINK : warning L4021: no stack segment
LINK : warning L4038: program has no starting address

C>
```

Even though the linker warns you that there is no stack segment, you don't need one right now. After you learn how to add more of the trappings, you will see why you might want a stack segment. You also can ignore the warning about no starting address, which is not important here since you will be creating a COM file (you will see why later).

Now you have your EXE file, but this still isn't the last step. Next, you need to create a COM version, which is just what you created with Debug. Later you will see why you need all of these steps. For now, you will create a COM version of Writestr.

For your final step, you will use a program EXE2BIN.EXE. Exe2bin, as its name implies, converts an EXE file to a COM, or binary (bin) file. There is a difference between EXE and COM files, but we won't deal with the differences until Chapter 11. For now just create the COM file by typing the following:

```
C>EXE2BIN WRITESTR WRITESTR.COM

C>
```

The response did not tell you very much. To see whether Exe2bin worked, list all of the Writestr files you have created so far.

```
C>DIR WRITESTR.*

 Volume in drive C has no label
 Volume Serial Number is 191C-8737
 Directory of C:\SOURCE\ASM

WRITESTR ASM        73 10-13-92   1:51p
WRITESTR OBJ       107 10-13-92   2:04p
WRITESTR EXE       520 10-13-92   2:07p
WRITESTR COM         8 10-13-92   2:12p
        4 file(s)        708 bytes
                     1179648 bytes free

C>
```

Type *writestr* to run the COM version and verify that your program functions properly (recall that it should print an asterisk on your screen). The exact sizes DOS reports for the first three files may vary a bit.

The results may seem a little anticlimactic, because it looks as though you are back where you were in Chapter 3. You are not; you have gained a great deal. It will become much clearer when you deal with calls again. Notice that you never once had to worry about where your program was put in memory, as you did about IP in Debug. The addresses were all taken care of for you.

Very soon you will appreciate this feature of the assembler: it will make programming much easier. For example, recall that in the last chapter you wasted space by placing your main program at 100h and the procedure you called at 200h. You will see that using the assembler allows us to place the procedure immediately after the main program without any gap. First, let's see how your program looks to Debug.

Looking at Writestr in Debug

To see how Debug reconstructs your program from the machine code of WRITESTR.COM, read your COM file into Debug and unassemble it, as follows:

```
C>DEBUG WRITESTR.COM
-U
397F:0100 B402          MOV     AH,02
397F:0102 B22A          MOV     DL,2A
397F:0104 CD21          INT     21
397F:0106 CD20          INT     20
     .
     .
     .
```

This is exactly what you had in Chapter 3. This is all Debug sees in WRITESTR.COM. The END and the two lines at the start of your source file did not make it through at all. What happened to them?

These directives do not appear in the final machine-language version of the program because they are for bookkeeping only. The assembler takes care of a lot of bookkeeping at the cost of some extra lines. You will make good use of directives to simplify your job. You will see how they affect your program when you learn about segments in Chapter 11.

Using Comments

Because you are no longer operating directly with Debug, you are free to add more to your program that the assembler sees but won't pass on to the 80x86. Perhaps the most important additions you can make are comments, which are invaluable in making a program clear. In assembly language programs, you place comments after a semicolon, which works like a single quotation mark (') in BASIC or the // in C++. The assembler ignores anything on the line after a semicolon, so you can add anything you want. If you add comments to your brief program you will see quite an improvement—you can understand this program without having to think back and remember what each line means.

```
.MODEL  SMALL
.CODE

        MOV     AH,2h           ;Select DOS function 2, character output
        MOV     DL,2Ah          ;Load the ASCII code for '*' to be printed
        INT     21h             ;Print it with INT 21h
        INT     20h             ;And exit to DOS

        END
```

Many programmers believe that well-written code doesn't need comments because well-written code is easy to understand. Not! While you may be able to understand what some code does without comments, you may not be able to figure out why *the code is doing what it is. Good comments should therefore say why your code is doing something, not just what it's doing.*

Using Labels for Code

To round off this chapter, let's take a look at labels, which are another bookkeeping feature of the assembler that make programming smoother.

Until now, you had to give a specific address when jumping from one part of a program to another with one of the jump commands. In everyday programming, inserting new instructions forces you to change the addresses in jump instructions. The assembler takes care of this problem with *labels*—names you give to the addresses of any instructions or memory locations. A label takes the place of an address. As soon as the assembler sees a label, it replaces the label with the correct address before sending it on to the 80x86, as in Figure 8-2.

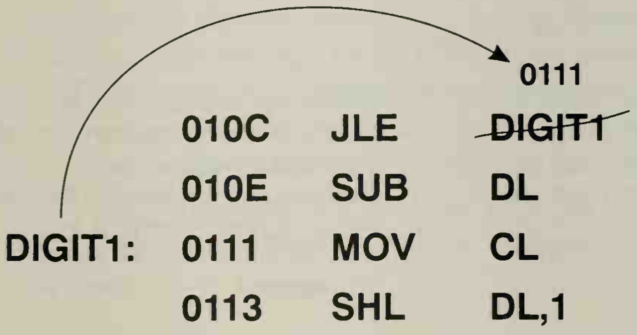


Figure 8-2: The assembler substitutes addresses for labels.

Labels can be up to 31 characters long and can contain letters, numbers, and any of the following symbols: a question mark (?), a period (.), an *at* symbol (@), an underline (_), or a dollar sign ($). They cannot start with a digit (0 through 9), and a period can be used only as the first character.

As a practical example, let's take a look at your program from Chapter 6 that reads a two-digit hex number. It contains two jumps, JLE 0111 and JLE 011F. The old version is as follows:

```
3985:0100 B401          MOV     AH,01
3985:0102 CD21          INT     21
3985:0104 88C2          MOV     DL,AL
3985:0106 80EA30        SUB     DL,30
3985:0109 80FA09        CMP     DL,09
3985:010C 7E03          JLE     0111
3985:010E 80EA07        SUB     DL,07
3985:0111 B104          MOV     CL,04
3985:0113 D2E2          SHL     DL,CL
3985:0115 CD21          INT     21
3985:0117 2C30          SUB     AL,30
3985:0119 3C09          CMP     AL,09
3985:011B 7E02          JLE     011F
3985:011D 2C07          SUB     AL,07
3985:011F 00C2          ADD     DL,AL
3985:0121 CD20          INT     20
```

The function of this program is not obvious. If this program is not fresh in your mind, it may take a while for you to understand it again. Adding comments and labels makes this program much easier to understand as follows:

```
.MODEL  SMALL
.CODE

        MOV     AH,1h           ;Select DOS function 1, character input
        INT     21h             ;Read a character, and return ASCII code in AL
        MOV     DL,AL           ;Move ASCII code into DL
        SUB     DL,30h          ;Subtract 30h to convert digit to 0 - 9
        CMP     DL,9h           ;Was it a digit between 0 and 9?
        JLE     DIGIT1          ;Yes, we have the first digit (four bits)
        SUB     DL,7h           ;No, subtract 7h to convert letter A - F
DIGIT1:
        MOV     CL,4h           ;Prepare to multiply by 16
        SHL     DL,CL           ;Multiply by shifting, becomes upper four bits
        INT     21h                 ;Get next character
        SUB     AL,30h              ;Repeat conversion
        CMP     AL,9h               ;Is it a digit 0 - 9?
        JLE     DIGIT2              ;Yes, so we have the second digit
        SUB     AL,7h               ;No, subtract 7h
DIGIT2:
        ADD     DL,AL               ;ADD second digit
        INT     20h                 ;And exit

        END
```

The labels here, DIGIT1 and DIGIT2, are of a type known as *NEAR* labels, because a colon (:) appears after the labels when they are defined. The term *NEAR* has to do with segments, which we will cover in Chapter 11 along with the MODEL, and CODE directives. If you assembled the preceding program and then unassembled it with Debug, you would see DIGIT1 replaced by 0111h and DIGIT2 replaced by 011Fh.

Summary

This has been quite a chapter. In a way, it is as if you have stepped into a new world. The assembler is much easier to work with than Debug, so you can now begin to write real programs because the assembler does much of the book-keeping for you.

What have you learned here? You began by learning how to create a source file and then you went through the steps of assembling, linking, and converting it from an OBJ file to an EXE, and then a COM file, using a simple program from Chapter 3. The assembly language program you created contained a few directives that you have never seen before, but which will become familiar once you have become more comfortable using the assembler. In fact, you will place MODEL, CODE, and END directives in all of your programs from now on, because they are needed, even though it will not be apparent why until Chapter 11.

Next, you learned about comments. You may have wondered how you survived without comments. Comments add so much to the readability of programs that you should not skimp on them.

Finally you learned about labels, which make our programs even more readable. You will use all of these ideas and methods throughout the rest of this book. Let's move on to the next chapter to see how the assembler makes procedures easier to use.

CHAPTER

Writing Procedures in Assembly Language

In this chapter you will learn how to write procedures in assembly language. This is much easier than writing them in machine language. You will also build the WRITE_CHAR and WRITE_HEX procedures, which you will use to build the Dskpatch program.

Files altered: VIDEO_IO.ASM

Disk file: VIDEO.9ASM

9

Topics Covered

The Assembler's Procedures

The Hex-Output Procedures

The Beginnings of Modular Design

A Program Skeleton

Summary

Now that you have met the assembler, it is time to become more comfortable writing assembly language programs. In this chapter, we will return to the subject of procedures. You will see how to write procedures much more easily with the help of the hard-working assembler. Then, you will move on to building some useful procedures, which you will use when you develop your Dskpatch program in later chapters.

You will begin with two procedures to print a byte in hexadecimal. Along the way, you will meet several more directives. But, like MODEL, CODE, and END in Chapter 8, they will be left mostly undefined until Chapter 11, where you will learn more about segments.

The Assembler's Procedures

When we first mentioned procedures, we left a large gap between the main program and its procedures so that we would have room for changes without having to worry about our main program overlapping a procedure. Now you have the assembler. Because it does all the work of assigning addresses to instructions, you no longer need to leave a gap between procedures. With the assembler, each time you make a change, you just assemble the program again.

You built a small program with one CALL in Chapter 7. The program did nothing more than print the letters A through J. It appeared as follows:

```
3985:0100 B241          MOV     DL,41
3985:0102 B90A00        MOV     CX,000A
3985:0105 E8F800        CALL    0200
3985:0108 FEC2          INC     DL
3985:010A E2F9          LOOP    0105
3985:010C CD20          INT     20

3985:0200 B402          MOV     AH,02
3985:0202 CD21          INT     21
3985:0204 C3            RET
```

Let's turn this into a program for the assembler. It will be hard to read without labels and comments, so add those embellishments to make your program far more readable as shown in Figure 9-1.

Listing 9-1 The Program PRINTAJ.ASM

```
.MODEL  SMALL
.CODE

PRINT_A_J       PROC
        MOV     DL,'A'                  ;Start with the character A
        MOV     CX,10                   ;Print 10 characters, starting with A
PRINT_LOOP:
        CALL    WRITE_CHAR              ;Print character
        INC     DL                      ;Move to the next char in the alphabet
        LOOP    PRINT_LOOP              ;Continue for 10 characters
        INT     20h                     ;Return to DOS
PRINT_A_J       ENDP

WRITE_CHAR      PROC
        MOV     AH,2                    ;Set function code for character output
        INT     21h                     ;Print the character already in DL
        RET                             ;Return from this procedure
WRITE_CHAR      ENDP

        END     PRINT_A_J
```

There are two new directives here: PROC, and ENDP. PROC and ENDP are directives for defining procedures. As you can see, both the main program and the procedure that was at 200h are surrounded by matching pairs of the directives PROC and ENDP.

PROC defines the beginning of a procedure; ENDP defines the end. The label in front of each is the name you give to the procedure they define. Thus, in the main procedure, PRINT_A_J, you can replace your CALL 200 instruction with the more readable CALL WRITE_CHAR. Just insert the name of the procedure, and the assembler assigns the addresses without a gap between procedures.

Because you have two procedures, you need to tell the assembler which to use as the main procedure—where the 80x86 should start executing your program. The END directive takes care of this detail. By writing END PRINT_A_J, you tell the assembler that PRINT_A_J is the main procedure. Later, you will see that the main procedure can be anywhere. Right now, however, you are dealing with .COM files, and you will need to place the main procedure first in our source file.

Now you are ready to begin. If you haven't done so yet, enter the program into a file called PRINTAJ.ASM and generate the .COM version, by using the same steps you did in the last chapter. Remember that you need to use different syntax in place of ML if you are using the Turbo Assembler.

```
ML /C PRINTAJ.ASM
LINK PRINTAJ;
EXE2BIN PRINTAJ PRINTAJ.COM
```

If you encounter any error messages that you do not recognize, check that you have typed in the program correctly.

Then give Printaj a try. Make sure you have run Exe2bin *before* you run Printaj. Otherwise, you will end up running the EXE version of Printaj, which will crash when it encounters the INT 20h instruction for reasons you will see in Chapter 11.

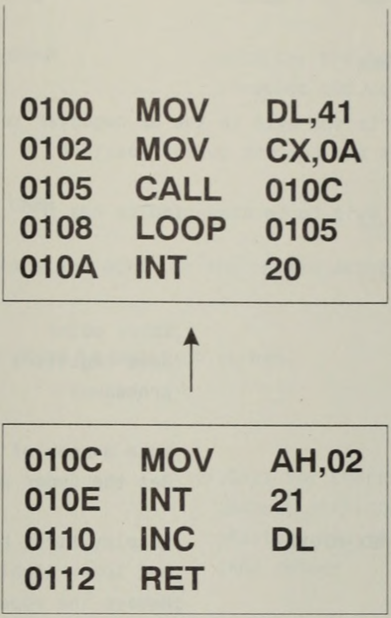
When you are satisfied, use Debug to unassemble the program and see how the assembler fits the two procedures together. Recall that you can read a particular file into Debug by typing its name as part of the command line. For example, type *DEBUG PRINTAJ.COM* and see the following:

```
-U
3985:0100 B241          MOV     DL,41
3985:0102 B90A00        MOV     CX,000A
3985:0105 E80600        CALL    010E
3985:0108 FEC2          INC     DL
3985:010A E2F9          LOOP    0105
3985:010C CD20          INT     20
3985:010E B402          MOV     AH,02
3985:0110 CD21          INT     21
3985:0112 C3            RET
```

The program is nice and snug, with no gap between the two procedures, as shown in Figure 9-1.

The Hex-Output Procedures

You have seen hex-output procedures twice before: once in Chapter 5, where you learned how to print a number in hex, and again in Chapter 7, where you saw how to simplify the program by using a procedure to print one hex digit. Now you are going to add yet another procedure to print one character.



```
0100   MOV    DL,41
0102   MOV    CX,0A
0105   CALL   010C
0108   LOOP   0105
010A   INT    20
```

```
010C   MOV    AH,02
010E   INT    21
0110   INC    DL
0112   RET
```

Figure 9-1: MASM assembles separate procedures without a gap.

By using a central procedure to write a character to the screen, you can change the way this procedure writes characters without affecting the rest of the program. It will be changed several times. Enter the following program into the file VIDEO_IO.ASM. The disk that accompanies this book includes the file under the name VIDEO_9.ASM as shown in Listing 9-2.

Listing 9-2 The New File VIDEO_IO.ASM (VIDEO_9.ASM)

```
.MODEL  SMALL
.CODE

TEST_WRITE_HEX  PROC
        MOV     DL,3Fh                  ;Test with 3Fh
        CALL    WRITE_HEX
```

continues

Listing 9-2 continued

```
        INT     20h                     ;Return to DOS
TEST_WRITE_HEX  ENDP

        PUBLIC  WRITE_HEX
;-----------------------------------------------------------------------;
; This procedure converts the byte in the DL register to hex and writes ;
; the two hex digits at the current cursor position.                    ;
;                                                                       ;
; On Entry:     DL      Byte to be converted to hex.                    ;
;                                                                       ;
; Uses: WRITE_HEX_DIGIT                                                 ;
;-----------------------------------------------------------------------;
WRITE_HEX       PROC                    ;Entry point
        PUSH    CX                      ;Save registers used in this
                                         procedure
        PUSH    DX
        MOV     DH,DL                   ;Make a copy of byte
        MOV     CX,4                    ;Get the upper nibble in DL
        SHR     DL,CL
        CALL    WRITE_HEX_DIGIT         ;Display first hex digit
        MOV     DL,DH                   ;Get lower nibble into DL
        AND     DL,0Fh                  ;Remove the upper nibble
        CALL    WRITE_HEX_DIGIT         ;Display second hex digit
        POP     DX
        POP     CX
        RET
WRITE_HEX       ENDP

        PUBLIC  WRITE_HEX_DIGIT
;----------------------------------------------------------------------;
; This procedure converts the lower 4 bits of DL to a hex digit and    ;
; writes it to the screen.                                             ;
;                                                                      ;
; On Entry:     DL      Lower 4 bits contain number to be printed      ;
;                        in hex.                                       ;
;                                                                      ;
; Uses:         WRITE_CHAR                                             ;
;----------------------------------------------------------------------;
WRITE_HEX_DIGIT PROC
        PUSH    DX                      ;Save registers used
        CMP     DL,10                   ;Is this nibble <10?
        JAE     HEX_LETTER              ;No, convert to a letter
        ADD     DL,"0"                  ;Yes, convert to a digit
        JMP     Short WRITE_DIGIT       ;Now write this character
HEX_LETTER:
        ADD     DL,"A"-10               ;Convert to hex letter
WRITE_DIGIT:
        CALL    WRITE_CHAR              ;Display the letter on the screen
        POP     DX                      ;Restore old value of DX
        RET
WRITE_HEX_DIGIT ENDP

        PUBLIC  WRITE_CHAR
;----------------------------------------------------------------------;
; This procedure prints a character on the screen using the DOS        ;
; function call.                                                       ;
;                                                                      ;
; On Entry:    DL      Byte to print on screen.                        ;
;----------------------------------------------------------------------;
WRITE_CHAR      PROC
        PUSH    AX
        MOV     AH,2                    ;Call for character output
        INT     21h                     ;Output character in DL register
        POP     AX                      ;Restore old value in AX
        RET                             ;And return
WRITE_CHAR      ENDP

        END     TEST_WRITE_HEX
```

The DOS function to print characters treats some characters specially. For example, using the DOS function to output 07 results in a beep, without printing the character for 07, which is a small diamond. You will see a new version of WRITE_CHAR that will print a diamond in Part III, where you will learn about the ROM BIOS routines inside your IBM PC. For now, we will just use the DOS function to print characters.

The new directive PUBLIC is here for future use. You will also use this directive in Chapter 13 when you learn about modular design. PUBLIC simply tells the assembler to generate some more information for the linker. The linker allows you to bring separate pieces of your program, assembled from different source files, together into one program. PUBLIC informs the assembler that the procedure named after the PUBLIC directive should be made public, or available to procedures in other files.

Right now, Video_io contains three procedures to write a byte as a hex number, as well as a short main program to test these procedures. You will be adding many procedures to the file as you develop Dskpatch. By the end of this book, VIDEO_IO.ASM will be filled with many general-purpose procedures.

The procedure TEST_WRITE_HEX that we have included does just what it says: it tests WRITE_HEX, which uses WRITE_HEX_DIGIT and WRITE_CHAR. As soon as you verify the accuracy of the function of these three procedures, you will remove TEST_WRITE_HEX from VIDEO_IO.ASM.

Create the COM version of Video_io, and use Debug to thoroughly test WRITE_HEX. Change the 3Fh at memory location 101h to each of the boundary conditions you tried in Chapter 5 (use the digits 0, 9, A, and F in different combinations for each of the two digits), then use G to run TEST_WRITE_HEX.

You will use many simple test programs to test new procedures that you have written. In this way, you can build a program piece by piece, rather than building and debugging it all at once. This incremental method is much faster and easier, because you confine bugs to the new code.

The Beginnings of Modular Design

Note that ahead of each procedure in Video_io we have included a block of comments briefly describing the function of each procedure. More importantly, these comments tell which registers the procedure uses to pass information back and forth, as well as what other procedures it uses. As one feature of your modular approach, the comment block allows you to use any procedure by looking at the description. There is no need to relearn how the procedure does its work. This also makes it fairly easy to rewrite one procedure without having to rewrite any of the procedures that call it.

We have also used PUSH and POP instructions to save and restore any registers used within each procedure. We will do this for every procedure we write, except for test procedures. This approach, too, is part of the modular style you will be using.

Recall that we save and restore any register used so we never have to worry about complex interactions between procedures trying to fight over the small number of registers in the 80x86. Each procedure is free to use as many registers as it likes, *provided* it restores them before the RET instruction. It is a small price to pay for the added simplicity. In addition, without saving and restoring registers, the task of rewriting procedures would be mind-rending. You would be sure to lose much hair in the process.

Also try to use many small procedures, instead of one large one. This makes the programming task simpler, although sometimes we will write longer procedures when the design becomes particularly convoluted.

These ideas and methods will all be borne out more fully in the chapters to come. In the next chapter, for example, we will add another procedure to Video_io: a procedure to take a word in the DX register and print the number in decimal on the screen.

A Program Skeleton

As you have seen in this and the preceding chapter, the assembler imposes a certain amount of overhead on any programs you write. In other words, you need to write a few directives that tell the assembler the basics. For future reference, the absolute minimum you will need for programs you write is as follows:

```
.MODEL  SMALL
.CODE

Some_procedure  PROC
        .
        .
        .
        INT     20h
Some_procedure  ENDP

        END     Some_procedure
```

We will add some new directives to this program skeleton in later chapters. You can use it, as shown here, as the starting point for new programs you write. Or, you can use some of the programs and procedures from this book as your starting point.

Summary

You are really making progress now. In this chapter, you learned how to write procedures in assembly language. From now on we will use procedures all the time. By using small procedures, you will make your programs more manageable.

You saw that a procedure begins with a PROC definition and ends with an ENDP directive. We rewrote PRINT_A_J to test your new knowledge of procedures, then went on to rewrite our program to write a hex number—this time with an extra procedure. Now that procedures are so easy to work with, there is little reason not to break programs into more procedures. In fact, you have seen ample reasons in favor of using many small procedures.

At the end of this chapter we talked briefly about modular design, a philosophy that will save you a great deal of time and effort. Modular programs will be easier to write and easier to read. They will also be easier for someone else to modify than programs created with the well-worn technique of spaghetti logic—programs written with very long procedures and many interactions.

Now you are ready to build another useful procedure. Then, in Chapter 11, you will learn about segments. From there you will move on to developing larger programs where you will really start to use the techniques of modular design.

CHAPTER

Printing in Decimal

In this chapter you will build a subroutine to display a number in decimal notation. You will use this subroutine later in the Dskpatch program. You will also learn about a few tricks that assembly-language programmers often use in their programs, mostly just for the fun of it. You will learn about the XOR (exclusive or) and the OR instructions, as well as the SI and DI registers.

Files altered: VIDEO_IO.ASM

Disk file: VIDEO_10.ASM

10

Topics Covered

Reviewing the Conversion to Decimal

Some Tricks and Shortcuts

The Inner Workings of WRITE_DECIMAL

Summary

We have been promising that we would write a procedure to take a word and print it in decimal notation. WRITE_DECIMAL uses some new tricks—ways to save a byte here and a few microseconds there. Perhaps such tricks will hardly seem to be worth the effort. But if you memorize them, you will find you can use them to shorten and speed up programs. Through these tricks, you will also learn about two new types of logical operations to add to the AND instruction covered in Chapter 5. First, let's review the process for converting a word to decimal digits.

Reviewing the Conversion to Decimal

Division is the key to converting a word to decimal digits. Recall that the DIV instruction calculates both the integer answer and its remainder. Calculating 12345/10 yields 1234 as the integer answer, and 5 as the remainder. In this example, 5 is simply the rightmost digit. If you divide by 10 again, you will get the next digit to the left. Repeated division by 10 *strips off* the digits from right to left, each time putting a digit in the remainder.

The digits come out in reverse order, but in assembly-language programming, there is a fix for that. Remember the stack? It is just like a stack of lunch trays: The first one to come off the top is the last tray that was set down. If you substitute digits for trays, and place the digits one on top of the other as they come out of the remainder, you will have it. You can pull out the digits in correct order.

The top digit is the first digit in your number and the other digits are underneath it. So, if you push the remainders as you calculate them and print the remainders as you pop them off the stack, the digits will be in correct order, as shown in Figure 10-1.

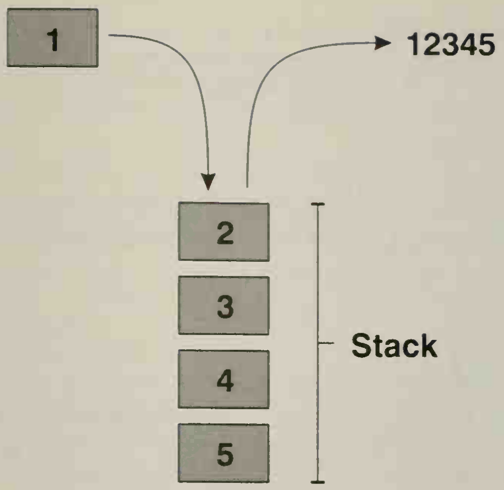


Figure 10-1: PUSHing the Digits onto the Stack Reverses Their Order.

The following program is the complete procedure to print a number in decimal notation. As mentioned, there are a few tricks hiding in this procedure. You will get to them soon enough, but let's try WRITE_DECIMAL to see if it works.

Place WRITE_DECIMAL into VIDEO_IO.ASM (see Listing 10-10), along with the proce-dures for writing a byte in hex. Make sure you place WRITE_DECIMAL *after* TEST_WRITE_HEX, which you will be replacing with TEST_WRITE_DECIMAL. To save some work, WRITE_DECIMAL uses WRITE_HEX_DIGIT to convert one nibble (four bits) into a digit.

Listing 10-1 Add to VIDEO_IO.ASM (Complete listing in VIDEO_10.ASM; see also Listing 10.2)

```
        PUBLIC  WRITE_DECIMAL
;-----------------------------------------------------------------------;
; This procedure writes a 16-bit, unsigned number in decimal notation.  ;
;                                                                       ;
```

continues

Listing 10-1 continued

```
; On Entry:     DX      N : 16-bit, unsigned number.                   ;
;                                                                      ;
; Uses:         WRITE_HEX_DIGIT                                        ;
;----------------------------------------------------------------------;
WRITE_DECIMAL   PROC
        PUSH    AX                      ;Save registers used here
        PUSH    CX
        PUSH    DX
        PUSH    SI
        MOV     AX,DX
        MOV     SI,10                   ;Will divide by 10 using SI
        XOR     CX,CX                   ;Count of digits placed on stack
NON_ZERO:
        XOR     DX,DX                   ;Set upper word of N to 0
        DIV     SI                      ;Calculate N/10 and (N mod 10)
        PUSH    DX                      ;Push one digit onto the stack
        INC     CX                      ;One more digit added
        OR      AX,AX                   ;N = 0 yet?
        JNE     NON_ZERO                ;Nope, continue
WRITE_DIGIT_LOOP:
        POP     DX                      ;Get the digits in reverse order
        CALL    WRITE_HEX_DIGIT
        LOOP    WRITE_DIGIT_LOOP
END_DECIMAL:
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
WRITE_DECIMAL   ENDP
```

Notice that we have included a new register, the SI (*Source Index*) register. Later you will see why it has been given that name. You will also meet its brother, the DI, or *Destination Index* register. Both registers have special uses, but they can also be used as if they were general-purpose registers. Since WRITE_DECIMAL needs four general-purpose registers, we used SI, even though we could have used BX, simply to show that SI (and DI) can serve as general-purpose registers if need be.

Before you try out our new procedure, you need to make two other changes to VIDEO_IO.ASM. First, remove the procedure TEST_WRITE_HEX, and insert the test procedure, which follows, in its place as shown in Listing 10-2.

Listing 10-2 Replace TEST_WRITE_HEX in VIDEO_IO.ASM with This Procedure (Complete listing in VIDEO_10.ASM)

```
TEST_WRITE_DECIMAL      PROC
        MOV     DX,12345
        CALL    WRITE_DECIMAL
        INT     20h                     ;Return to DOS
TEST_WRITE_DECIMAL      ENDP
```

This procedure tests WRITE_DECIMAL with the number 12345 (which the assembler converts to the word 3039h).

Second, you need to change the END statement at the end of VIDEO_IO.ASM to read END TEST_WRITE_DECIMAL, because TEST_WRITE_DECIMAL is now your main procedure.

Make these changes and give VIDEO_IO a whirl. Convert it to its .COM version and see if it works. If it doesn't, check your source file for errors. If you are adventurous, try to find your bug with Debug.

Some Tricks and Shortcuts

Hiding in WRITE_DECIMAL are two tricks of the trade garnered from the people who wrote the ROM BIOS procedures you will meet in Chapter 17. (IBM used to print the source code for their ROM BIOS, but that kind of sharing is a thing of the past.) The first is an efficient instruction to set a register to zero. It is not much more efficient than MOV AX,0, and perhaps it is not worth the effort, but it is the sort of trick you will find people using, so here it is. The following instruction sets the AX register to zero.

```
XOR        AX,AX
```

How? To understand that, we need to learn about the logical operation called an *Exclusive OR*, hence the name XOR.

The exclusive OR is similar to an OR (which you will see next), but the result of XORing two trues is true if *only* one bit is true, not if both are true.

XOR	0	1
0	0	1
1	1	0

Thus, if you exclusive OR a number to itself, you get zero:

```
    1 0 1 0  0 1 0 1
XOR 1 0 1 1  0 1 0 1
    0 0 0 0  0 0 0 0
```

That is the trick. You won't find other uses for the XOR instruction in this book, but we thought you would find it interesting.

As a short aside, you will also find many people using another quick trick to set a register to zero. Rather than using the XOR instruction, we could have used the following to set the AX register to zero:

```
SUB        AX,AX
```

Now for the other trick. It is just about as devious as our XOR scheme to clear a register, and it uses a cousin to the exclusive OR—the OR function.

We want to check the AX register to see if it is zero. To do this, we could use the instruction CMP AX,0. But we would rather use a trick: It's more fun, and a little more efficient, too. So, we write OR AX,AX and follow this instruction with a JNE (Jump if Not Equal) conditional jump. (We could also have used JNZ—Jump if Not Zero.)

The OR instruction, like any of the math instructions, sets the flags, including the zero flag. Like AND, OR is a logical concept. But here, a result is true if one *OR* the other bit is true.

OR	0	1
0	0	1
1	1	1

If we take a number and OR it to itself, you get the original number back again:

```
    1011 0101
OR  1011 0101
    1011 0101
```

The OR instruction is also useful for setting just one bit in a byte. For example, we can set bit 3 in the number we just used:

```
    1011 0101
OR  0000 1000
    1011 1101
```

There will be more tricks to play before you are through with this book, but these two are the only ones that are entirely for fun.

The Inner Workings of WRITE_DECIMAL

To see how WRITE_DECIMAL performs its task, study the listing; we won't cover many details here. However, we do need to point out a few more things.

First, the CX register is used to count how many digits you pushed onto the stack, so you know how many to remove. The CX register is a particularly convenient choice, because you can build a loop with the LOOP instruction and use the CX register to store the repeat count. This choice makes the digit-output loop (WRITE_DIGIT_LOOP) almost trivial, because the LOOP instruction uses the CX register directly. You will use CX often when you have to store a count.

Next, be careful to check the boundary conditions here. The boundary condition at 0 isn't a problem, as you can check. The other boundary condition is 65535, or FFFFh, which you can check easily with Debug. Just load VIDEO_IO.COM into Debug by typing *DEBUG VIDEO_IO.COM* and change the 12345 (3039h) at 101h to 65535 (FFFFh). (WRITE_DECIMAL works with unsigned numbers. See if you can write a version to display signed numbers.)

You may have noticed a sticky point here, having to do with the 80x86, not your program. Debug works mostly with bytes (at least the E command does), but you want to change a word. You must be careful, since the 80x86 stores the bytes in a different order. An unassemble for the MOV instruction is as follows:

```
3985:0100 BA3930        MOV     DX,3039
```

You can tell from the *BA3930* part of this display that the byte at 101h is 39h, and the one at 102h is 30h (BA is the MOV instruction). These two bytes are the two bytes of 3039h, but seemingly in reverse order. Confusing? The order is logical, after a short explanation.

A word consists of two parts, the lower byte and the upper byte. The lower byte is the least significant byte (39h in 3039h), while the upper byte is the most significant byte (30h) as you can see in Figure 10-2. It makes sense, then, to place the lower byte at the lower address in memory. (Many other computer architectures, such as the Motorola 680x0 family used in the Apple Macintosh, actually reverse these two bytes, and this can be a bit confusing if you are writing programs on several different types of computers.)

Try different numbers for the word starting at 101h, and you will see how this storage works. Use TEST_WRITE_DECIMAL to see if you got it right, or unassemble the first instruction.

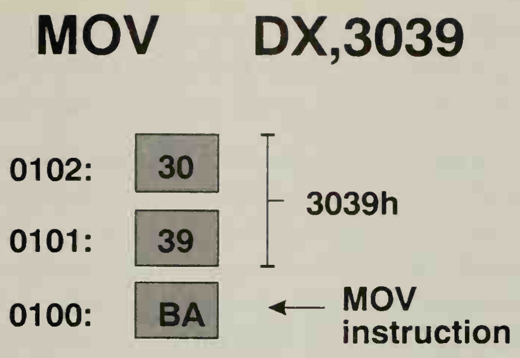


Figure 10-2: The 80x86 Stores Numbers with the Lower Byte First in Memory.

Summary

You added a few new instructions to your repertoire, as well as a few tricks for fun. You also learned about two other registers, SI and DI, that you can use as general-purpose registers. They have other uses that you will see in later chapters.

You learned about the XOR and OR logical instructions, which allow you to work between individual bits in two bytes or words. In your WRITE_DECIMAL procedure, you used the XOR AX,AX instruction as a tricky way to set the AX register to zero. You used OR AX,AX as a devious way to write the equivalent of CMP AX,0 to test the AX register and see if it is zero. Finally, you learned about how the 80x86 stores a word in memory by checking the boundary conditions of your new procedure, WRITE_DECIMAL.

You now have another general-purpose procedure, WRITE_DECIMAL, that you will be able to use in the future for your own programs.

Take a breather now. We've got a few *different* chapters scheduled next. Chapter 11 covers segments in detail. Segments are perhaps the most complicated part of the 80x86 microprocessor, so the chapter may prove to be rather heavy going. Even so, we need to cover the topic for following chapters.

After that, you will make a slight course correction and get back on track by learning what to do with the program Dskpatch. You will do a bit of probing on disks, and learn about sectors, tracks, and other such things.

From there, we can plot a simple course for preliminary versions of Dskpatch. En route, you will get a chance to see how to develop large programs. Programmers don't write an entire program, then debug it. They write sections and try each section before they move on—programming is much less work that way. You have used this approach to a limited extent by writing and testing WRITE_HEX and WRITE_DECIMAL, for which the test programs were very simple. The test programs from here on will be more complex, but more interesting, too.

C H A P T E R

Segments

In this chapter you will learn about segments, which are very important to assembly-language programs. From here on, you will always build EXE, rather than COM, programs. These programs use at least two segments. You will also learn about the PSP (Program Segment Prefix), about the DOSSEG directive for controlling segment order, about NEAR and FAR calls for working with large programs, and more about how INT instructions work.

Files altered: None

11

Topics Covered

How Memory Is Divided into Segments

The Stack

The Program Segment Prefix (PSP)

The DOSSEG Directive

NEAR and FAR CALLs

More on the INT Instruction

Interrupt Vectors

Summary

In the preceding chapters, you encountered several directives that dealt with segments. Now the time has come to look at segments themselves, and at how the 80x86 manages to address a full megabyte (1,048,576 bytes) of memory under DOS. From this, you will begin to understand why segments need their own directives in the assembler, and in later chapters you will begin to use different segments (thus far, you have used only one).

Let's start at the 80x86 level by learning how it constructs the 20-bit addresses needed for a full megabyte of memory.

How Memory Is Divided into Segments

Segments are about the only part of the 80x86 that you have not covered yet. They are, perhaps, the most confusing part of this microprocessor. In fact, segments are what you call a *kludge* in this business—computerese for a makeshift fix to a problem. The 80386 and 80486 microprocessors have additional addressing modes that are much simpler and do not use segments. Unfortunately DOS does not use this mode (the WIN32s programming kit for Windows does allow you to use this *linear* addressing mode).

The problem, in this case, is being able to address more than 64K of memory—the limit with one word, since 65535 is the largest number a single word can hold. Intel, designers of the 80x86, used segments and segment registers to "fix" this problem, and in the process made the 80x86 more confusing.

So far, you have not had to worry about this problem. You have been using the IP register to hold the address of the next instruction for the 80x86 to execute ever since you met Debug in Chapter 2. Back then, you may recall we said the address is actually formed from both the CS register and the IP register. But we never really said how. Let's find out.

Although the complete address is formed from two registers, the 80x86 does not form a two-word number for the address. If you were to take CS:IP as a 32-bit number (two 16-bit numbers side by side), the 80x86 would be able to address about four billion bytes—far more than the one million bytes it can actually address in DOS. The 80x86's method is slightly more complicated. The CS register provides the *starting* address for the code segment, where a segment is 64K of memory.

As you can see in Figure 11-1, the 80x86 divides memory into many overlapping segments, with a new segment starting every 16 bytes. The first segment (segment 0) starts at memory location 0; the second (segment 1) starts at 10h (16); the third starts at 20h (32), and so on.

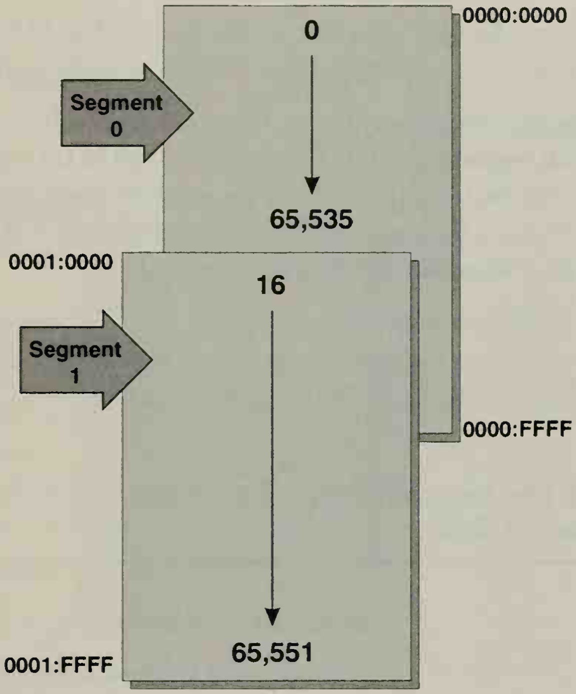


Figure 11-1: Overlapping segments start every 16 bytes and are 65536 bytes long.

The actual address is just CS * 16 + IP. For example, if the CS register contains 3FA8 and IP contains D017, the absolute address is evident as follows.

```
  CS * 16 : 0 0 1 1 1 1 1 1 1 0 1 0 1 0 0 0 0 0 0
+ IP      : 1 1 0 1 0 0 0 0 0 0 0 1 0 1 1 1     1
-------------------------------------------------
            0 1 0 0 1 1 0 0 1 0 1 0 1 0 0 1 0 1 1 1
```

We multiplied by 16 by shifting CS left four bits and injecting zeros at the right, as you can see in Figure 11-2.

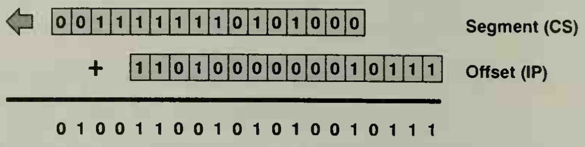


*Figure 11-2: The absolute address of CS:IP is CS * 16 + IP.*

Now, this may seem like a strange way to address more than 64K of memory—but it works. Soon, you will begin to see how well it really works.

The 80x86 has four segment registers: CS (Code Segment), DS (Data Segment), SS (Stack Segment), and ES (Extra Segment). The CS register you have been looking at is used by the 80x86 for the segment where the next instruction is stored. In much the same way, DS is the segment where the 80x86 looks for data, and SS is where the 80x86 places the stack.

Before we go on, let's look at a short program, which is quite different from any you have seen before. This short program uses two different segments. Enter this program into the file TEST_SEG.ASM as follows:

Listing 11-1 The Program TEST_SEG.ASM (Not included on disk)

```
DOSSEG
.MODEL SMALL

.STACK                                      ;Allocate a 1K stack

.CODE

TEST_SEGMENT    PROC
        MOV     AH,4Ch                      ;Ask for the exit-to-dos function
        INT     21h                         ;Return to DOS
TEST_SEGMENT    ENDP

        END     TEST_SEGMENT
```

Assemble and link Test_seg, but do not generate a COM file for it. (If you are using MASM 6 or later, you can type "ML TEST_SEG.ASM" to both assemble and link in one step!) The result will be TEST_SEG.EXE, which is slightly different from a COM file.

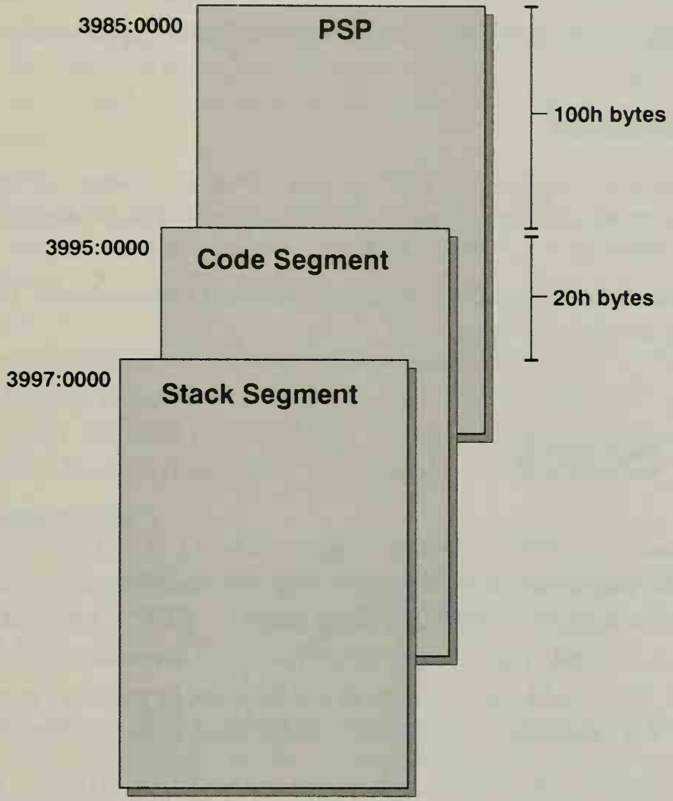


Figure 11-3: Memory layout for TEST_SEG.EXE.

Note

You have to use a method other than INT 20h in order to exit from EXE files. For COM files, INT 20h works perfectly well, but it doesn't work at all for EXE files because the organization of segments is very different, as you will see in this chapter; more on this difference later. From now on you will use INT 21h, function 4Ch to exit programs.

When you use Debug on a COM file, Debug sets all the segment registers to the same number, with the program starting at an *offset* of 100h from the start of this segment. The first 256 bytes (100h) are used to store various pieces of information. Although this area does not contain much you will need, we will look at part of this area soon.

Try loading TEST_SEG.EXE into Debug to see what happens with segments in an EXE file.

```
C>DEBUG TEST_SEG.EXE
-R
AX=0000  BX=0000  CX=0004  DX=0000  SP=0400  BP=0000  SI=0000  DI=0000
DS=3985  ES=3985  SS=3997  CS=3995  IP=0000   NV UP EI PL NZ NA PO NC
3995:0000 B44C          MOV     AH,4C
```

The values of the SS and CS registers are different from those for DS and ES as you can also see in Figure 11-3.

The Stack

Two segments are defined in this program. The STACK segment (which is actually the data segment) is where you place the stack (hence, the .STACK), and the code segment (which is actually called _TEXT) is where all your instructions are stored. The .STACK directive tells the assembler to create a 1024 byte stack. (You could create a larger or smaller stack by putting a number after .STACK. For example, .STACK 128 would create a stack 128 bytes long.)

The address for the top of the stack is given by SS:SP. SP is the Stack Pointer, like IP is the instruction pointer for code, and is an offset within the current Stack Segment.

Actually, "top-of-stack" is a misnomer, because the stack grows from high memory toward low memory. Therefore, the *top* of the stack is really at the bottom of the stack in memory, and new entries to the stack are placed progressively lower in memory. Here, SP is 400h, which is 1024 decimal, because you defined a stack area 1024 bytes long. You haven't placed anything on the stack as yet, so top-of-stack is still at the top of the memory you set aside for the stack: 400h.

If you think back to the COM programs in previous chapters, you never declared a stack segment, which raises two questions: Why didn't you have to declare a stack segment for COM programs? And where was the stack in the COM programs? All the COM programs you created had only one segment, and all the segment registers (CS, DS, ES, and SS) pointed to this segment. Since you had just one segment, you didn't need a separate stack segment. As to where the stack was, if you look at the register display for WRITESTR.COM, you will see the stack is at the very end of the segment (SP = FFEE). The stack is as follows:

```
-R
AX=0000  BX=0000  CX=0000  DX=0000  SP=FFEE  BP=0000  SI=0000  DI=0000
DS=3995  ES=3995  SS=3995  CS=3995  IP=0100   NV UP EI PL NZ NA PO NC
3995:0100 B402          MOV     AH,02
-
```

DOS always sets the stack pointer to the very end of the segment when it loads a COM file into memory. For this reason, you do not need to declare a stack segment (with .STACK) for COM files. What would happen if you removed the .STACK directive from TEST_SEG.ASM?

```
C>DEBUG TEST_SEG.EXE
-R
AX=0000  BX=0000  CX=0004  DX=0000  SP=0000  BP=0000  SI=0000  DI=0000
DS=3985  ES=3985  SS=3995  CS=3995  IP=0000   NV UP EI PL NZ NA PO NC
3D90:0000 B44C          MOV     AH,4C
-
```

The stack is now at 3995:0, which is the start of your program (CS:0). This is very bad news. You do not want the stack anywhere near your program's code. Since the stack pointer is at SS:0, it has no room to grow (since the stack grows down in memory). For these reasons, you *must* declare a stack segment for EXE programs.

Always declare a stack segment with .STACK in EXE programs.

Getting back to the two-segment example, notice that the Stack Segment (SS) is segment number 3997 (this will probably be different for you), while our Code Segment (CS) is at segment 3995—two less than SS, or just 32 bytes lower in memory. Since we did not put any data into the stack segment, unassembling starting at CS:0 will show our program (MOV AH,4C and INT 21) followed by whatever happened to be in memory.

```
-U CS:0
3995:0000 B44C          MOV     AH,4C
3995:0002 CD21          INT     21
3995:0004 65            DB      65
3995:0005 2028          AND     [BX+SI],CH
3995:0007 59            POP     CX
3995:0008 2F            DAS
3995:0009 4E            DEC     SI
3995:000A 293F          SUB     [BX],DI
       .
       .
       .
```

You will almost certainly see different instructions after the INT 21h in this Debug listing.

The Program Segment Prefix (PSP)

The "scratch area" is actually called a PSP (Program Segment Prefix) and contains information for use by DOS. In other words, do not assume you can make use of this area.

In looking at the register display, you may have noticed that the ES and DS registers contain 3985h, 10h less than the beginning of the program at segment 3995h. Multiplying by 16 to get the number of bytes you can see that there are 100h (or 256) bytes before your program starts. This is the same scratch area placed at the beginning of a COM file.

Among other things, this 256 byte PSP at the start of programs contains characters typed after the name of the program, starting at 80h from the start of the PSP. For example:

```
C>DEBUG TEST_SEG.EXE And now for some characters you will see in the memory dump
-D DS:80
3985:0080  39 20 41 6E 64 20 6E 6F-77 20 66 6F 72 20 73 6F   9 And now for so
3985:0090  6D 65 20 63 68 61 72 61-63 74 65 72 73 20 77 65   me characters we
3985:00A0  27 6C 6C 20 73 65 65 20-69 6E 20 74 68 65 20 6D   'll see in the m
3985:00B0  65 6D 6F 72 79 20 64 75-6D 70 0D 20 6D 65 6D 6F   emory dump. memo
3985:00C0  72 79 20 64 75 6D 70 0D-00 00 00 00 00 00 00 00   ry dump.........
       .
       .
       .
```

The first byte says you typed 39h (or 57) characters, including the first space after TEST_SEG.EXE. You won't use this information in this book, but it helps show why you might want such a large PSP.

The PSP also contains information that DOS uses when exiting from a program, with either the INT 20h or the INT 21h, function 4Ch instructions. For reasons that are not clear, the INT 20h instruction expects the CS register to point to the start of this PSP, which it does for a COM program but *not* for an EXE program. The exit function (INT 21h, function 4Ch) was added to DOS with the introduction of version 2.00 to make it easier to exit from EXE programs; function 4Ch does not expect the CS register to point to the start of the PSP. You should use INT 21h, function 4Ch from now on to exit programs.

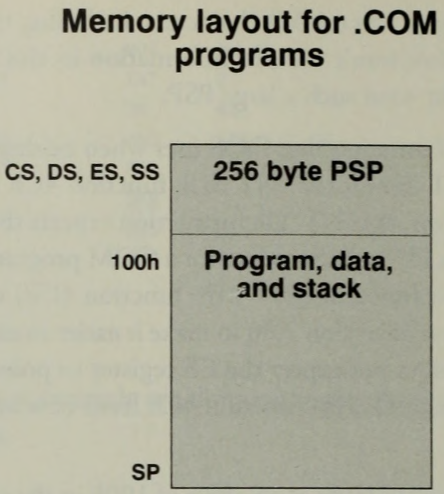
The code for COM files always starts at an offset of 100h in the code segment to leave room for this 256-byte PSP at the start. This is unlike the EXE file, which had its code start at IP = 0000, because the code segment started 100h bytes after the beginning of the area in memory.

In the early days of DOS, most programs were written as COM programs because they were slightly simpler to write. But today, almost all programs are written as EXE programs. So in the rest of this book, we will be working almost entirely with EXE programs. Figure 11-4 summarizes the differences between COM and EXE programs.

The DOSSEG Directive

If you take a look again at TEST_SEG.EXE, you will notice that the stack segment is higher in memory than the code segment. Yet in the source file you defined the stack (.STACK) *before* any of the code (.CODE). So why is the stack higher in memory than the code?

The DOSSEG directive at the start of the program tells the assembler that you want the segments of your program loaded in a very specific order with the code segment appearing first, and the stack last. In Chapter 14 you will see more about DOSSEG and the order of segments when you add another segment to hold data.



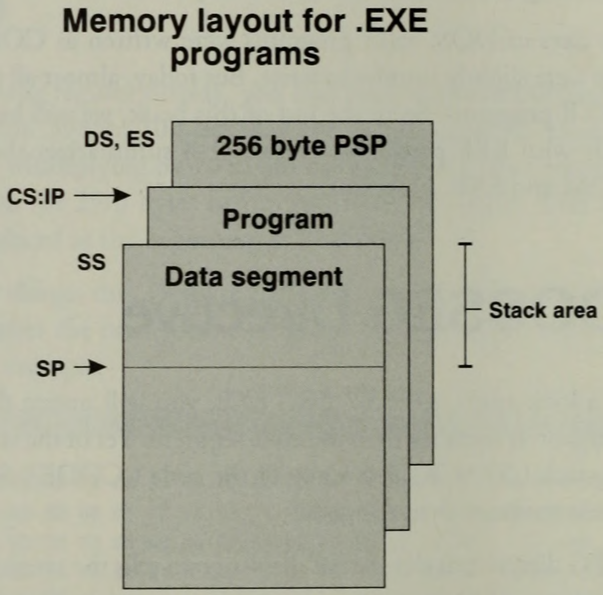


Figure 11-4: COM vs EXE programs.

NEAR and FAR CALLs

The rest of the information in this chapter is purely for your interest, since you won't be making use of it in this book. Skip the next two sections and read them later if you find the going tough or you are anxious to get back to programming.

Let's step back for a minute and take a closer look at the CALL instructions used in previous chapters. Specifically let's look at the short program in Chapter 7, where you first learned about the CALL instruction. You wrote a very short program that looked like the following (without the procedure at 200h):

```
3985:0100 B241          MOV     DL,41
3985:0102 B90A00        MOV     CX,000A
3985:0105 E8F800        CALL    0200
3985:0108 E2FB          LOOP    0105
3985:010A CD20          INT     20
```

As you can see by looking at the machine code on the left, the CALL instruction occupies only three bytes (E8F800). The first byte (E8h) is the CALL instruction and the second two bytes form an offset. The 80x86 calculates the address of the routine you are calling by adding this offset of 00F8h (remember that the 80x86 stores the lower byte of a word in memory *before* the high byte, so you have to reverse the bytes) to the address of the next instruction (108h in the program). In this case you have F8h + 108h = 200h.

The fact that this instruction uses a single word for the offset means that CALLs are limited to a single segment, which is 64K bytes long. So how is it that you can write a program like Lotus 1-2-3 that is larger than 64K? By using FAR, rather than NEAR, calls.

NEAR CALLs are limited to a single segment. In other words, they change the IP register without affecting the CS register. For this reason they are sometimes known as *intrasegment* CALLs.

You can also have FAR CALLs that change both the CS *and* IP registers. Such CALLs are often known as *intersegment* CALLs because they call procedures in other segments.

Going along with these two versions of the CALL instruction are two versions of the RET instruction. The NEAR CALL, as you saw in Chapter 7, pushes a single word onto the stack for its return address. The corresponding RET instruction pops this word off the stack and into the IP register.

In the case of FAR CALLs and RETs, a word is not sufficient because we're dealing with another segment. In other words, you need to save a two-word return address on the stack: one word for the instruction pointer (IP) and the other for the code segment (CS). The FAR RET pops two words off the stack—one for the CS register, and the other for IP.

How does the assembler know which of these two CALLs and RETs to use? When should it use the FAR CALL, and when should it use the NEAR CALL? By putting a NEAR or FAR directive after the PROC directive. By way of example, look at the following program:

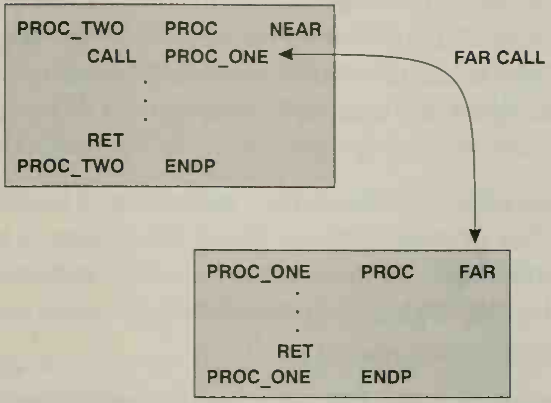
```
PROC_ONE        PROC    FAR
          .
          .
          .
        RET
PROC_ONE        ENDP


PROC_TWO        PROC    NEAR
        CALL    PROC_ONE
          .
          .
          .
        RET
PROC_TWO        ENDP
```

When the assembler sees the CALL PROC_ONE instruction, it hunts in its table for the definition of PROC_ONE, which, is PROC_ONE PROC FAR. This definition tells whether the procedure is a NEAR or FAR procedure.

In the case of a NEAR procedure, the assembler generates a NEAR CALL. And conversely, it generates a FAR CALL, as shown in Figure 11-5, if the procedure you're calling was defined as a FAR procedure. In other words, the assembler uses the definition of the procedure that you're *calling* to determine the type of CALL instruction that is needed.

For the RET instruction, the assembler looks at the definition of the procedure that contains the RET instruction. In your program, the RET instruction for PROC_ONE will be a FAR RET, as shown in Figure 11-6, because

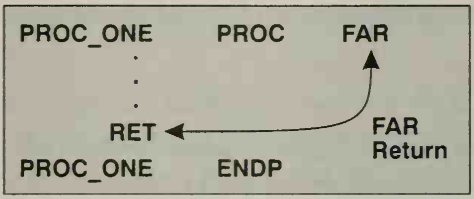
PROC_ONE is declared to be a FAR procedure. Likewise, the RET in PROC_TWO is a NEAR RET.



```
PROC_TWO    PROC      NEAR
      CALL  PROC_ONE
            .
            .
            .
      RET
PROC_TWO    ENDP
```

```
PROC_ONE      PROC     FAR
        .
        .
        .
       RET
PROC_ONE      ENDP
```

Figure 11-5: The assembler produces a FAR CALL.



```
PROC_ONE      PROC     FAR
       .
       .
       .
       RET
PROC_ONE      ENDP
```

Figure 11-6: The assembler produces a FAR RET.

What happens when you don't put a NEAR or FAR directive after the PROC? The assembler uses the information in the .MODEL directive to determine whether procedures are NEAR or FAR if you don't explicitly declare a procedure as NEAR or FAR. We're using the .MODEL SMALL directive, which tells the assembler that you have only one code segment, so all the procedures are NEAR procedures. There are other .MODEL directives (such as MEDIUM) that tell the assembler to make procedures FAR if they are not explicitly declared as NEAR.

More on the INT Instruction

The INT instruction is much like a CALL instruction, but with a minor difference. The name *INT* comes from the word *interrupt.* An interrupt is an external signal that causes the 80x86 to execute a procedure and then return to what it was doing before it received the interrupt. An INT instruction doesn't interrupt the 80x86, but it is treated as if it did.

When the 80x86 receives an interrupt, it needs to store more information on the stack than just the two words for the return address. It has to store the values of the status flags—the carry flag, the zero flag, and so on. These values are stored in the Flag Register, and the 80x86 pushes this information onto the stack before the return address.

Your IBM PC regularly responds to a number of different interrupts. The 80x86 inside your IBM PC receives an interrupt from the clock 18.2 times every second, for example. Each of these interrupts causes the 80x86 to stop what it is doing and execute a procedure to count the clock pulses. Now, envision such an interrupt occurring between the following two program instructions.

```
CMP     AH,2
JNE     NOT_2
```

Assume AH = 2, so the zero flag will be set after the CMP instruction, which means that the JNE instruction will not branch to NOT_2.

Now, imagine that the clock interrupts the 80x86 between these two instructions. That means the 80x86 runs off to carry out the interrupt procedure before it checks the zero flag (with the JNE instruction). If the 80x86 didn't save and restore the flag registers, the JNE instruction would use flags set by the interrupt procedure, *not* from your CMP instruction. To prevent such disasters, the 80x86 *always* saves and restores the flag register for interrupts. An interrupt saves the flags, and an IRET (*Interrupt Return*) instruction restores the flags at the end of the interrupt procedure. The same is true for an INT instruction. After executing the INT 21 instruction, the 80x86's stack will look like this:

```
Top of stack →  Old IP (return address part I)
                Old CS (return address part II)
                Old Flag Register
```

The stack grows into lower memory, so the Old Flag Register is actually highest in memory.

When you place an INT instruction in a program, the interrupt is no surprise. Why save the flags? Isn't saving the flags useful only when an external interrupt comes at an unpredictable time? As it turns out, the answer is no. There is a very good reason for saving and restoring the flags for INT instructions. In fact, without this feature, Debug would not be possible.

Debug uses a special flag in the flag register called the Trap Flag. This flag puts the 80x86 into a special mode known as *single-step* mode. Debug uses this to trace through programs one instruction at a time. When the trap flag is set, the 80x86 issues an INT 1 after it executes any instruction.

The INT 1 also clears the trap flag so the 80x86 won't be in single-step mode while you are inside Debug's INT 1 procedure. Since INT 1 saved the flags to the stack, issuing an IRET to return to the program you are debugging restores the trap flag. Then, you will receive another INT 1 interrupt after the next instruction in your program. This is just one example of when it is useful to save the flag registers. But, as you will see next, this restore-flag feature isn't always appropriate.

Some interrupt procedures bypass the restoration of the flag registers. For example, the INT 21h procedure in DOS sometimes changes the flag registers by short-circuiting the normal return process. Many of the INT 21h procedures that read or write disk information return with the carry flag set if there was an error of some sort (such as no disk in the drive).

Interrupt Vectors

Where do these interrupt instructions get the addresses for procedures? Each interrupt instruction has an interrupt number, such as the 21h in INT 21h. The 80x86 finds addresses for interrupt procedures in a table of *interrupt vectors*, which is located at the bottom of memory. For example, the two-word address for the INT 21h procedure is at 0000:0084. You get this address by multiplying the interrupt number by 4 (4 * 21h = 84h), because we need four bytes (two words) for each vector or procedure address.

Vectors are exceedingly useful for adding features to DOS because they enable you to intercept calls to interrupt procedures by changing the addresses in the vector table. We will use this trick at the end of this book to add a disk light to your computer's screen.

All of these ideas and methods should become clearer as you see more examples. Most of this book from here on will be filled with examples, so there will be plenty to study. If you have been feeling a bit overwhelmed by new information, rest easy. We will take a short breather in the next chapter to get reoriented and back on course.

Summary

This chapter contained a lot of information. You won't use it all, but you did need to learn more about segments. Chapter 13 covers modular design, and you will use some aspects of segments to make your job easier.

You began this chapter by learning how the 80x86 divides memory into segments. To understand segments in more detail, you built an EXE program with two different segments. You also learned that you need to use INT 21h, function 4Ch rather than INT 20h to exit from EXE programs. This is important since you will be using EXE programs from now on in this book.

You also found that the 100h (256-byte) PSP (Program Segment Prefix) at the start of your programs contains a copy of what you typed on the command line. You won't use this knowledge in this book, but it helps you see why DOS sets aside such a large chunk of memory for the purpose.

Finally you learned more about the DOSSEG, MODEL, .CODE, STACK, NEAR, and FAR directives. These are all directives that help you work with segments. In this book, you will barely use the power of these directives, because our EXE programs will use only two segments. But for programmers who write *very large* programs in assembly language (using the MEDIUM memory model), these directives are invaluable. If you are interested, you will find the details in your macro assembler manual.

At the very end of this chapter you learned more about the roots of the helpful INT instruction. Now, you are just about ready to slow down and learn how to *write* larger and more useful assembly language programs.

C H A P T E R

Course Corrections—How To Build Dskpatch

In this chapter you will learn more about what is covered in the rest of this book. In particular, you will look at the game plan you will use to build the Dskpatch program that will be the center of attention until Part IV.

Files altered: None

12

Topics Covered

We have been poking into a lot of new and interesting places. You may have wondered whether we have been wandering about somewhat aimlessly. We haven't been, of course. You are now familiar enough with your new surroundings for us to plot a course for the rest of this book. We will take a close look at a design for the Dskpatch program. Then you will spend the rest of this book developing Dskpatch, much as you will later develop programs of your own.

We will not present the finished version of Dskpatch all at once; that is not the way we wrote it. Instead, we will present short test programs to check each stage of your program as you write it. To do this, you need to know where you want to go. Hence, our course correction here. Since Dskpatch will deal with information on disks, that's where you will begin.

Disks, Sectors, and So Forth

The information on your floppy disks is divided into *sectors*. Each sector holds 512 bytes of information. A double-sided, double-density 5-1/4" disk formatted with DOS 2.0 or above has a total of 720 sectors, or 720 * 512 = 368,640 bytes (see Table 12-1 for other types of disks). If you could look directly at these sectors, you could examine the directory directly, or you could look at the files on the disk. You cannot—not by yourself—but Dskpatch will. Let's use Debug to learn more about sectors and get an idea of how you will display a sector with Dskpatch.

Debug has a command, L (*Load*), to read sectors from disk into memory where you can look at the data. As an example, let's look at the directory that starts at sector 5 on a double-sided disk (use Table 12-1 to determine what number to use for the directory if you have a different type of disk). Load sector 5 from the disk in drive A (drive 0 to Debug) by using the L command. Make sure you have a 360K (or 1.2M, 720K, or 1.44M) disk in drive A, then enter the following:

```
-L100 0 5 1
```

Table 12-1 Starting Sector for the Root Directory

Disk Type	*Sectors/disk*	*Directory*
5-1/4", 360K	720	5h
5-1/4", 1.2M	2,400	Fh
3-1/2", 720K	1,440	7h
3-1/2", 1.44M	2,880	13h

As you can see in Figure 12-1, this command loads sectors into memory, starting with sector 5 and continuing through one sector at an offset of 100 within the data segment.

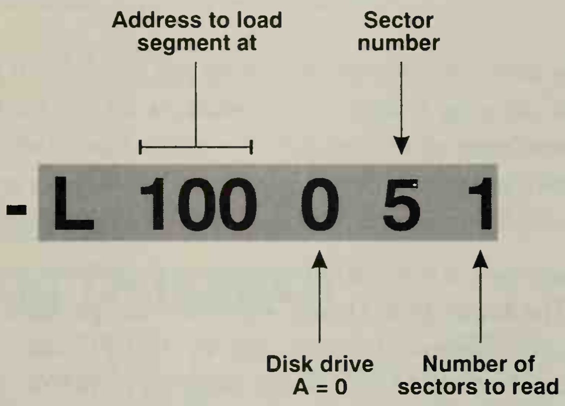


Figure 12-1: DEBUG's Load Command.

To display sector 5, you can use a Dump command as follows:

```
-D 100
396F:0100  49 42 4D 42 49 4F 20 20-43 4F 4D 27 00 00 00 00   IBMBIO  COM'....
396F:0110  00 00 00 00 00 00 00 60-68 06 02 00 00 12 00 00   .......'h.......
396F:0120  49 42 4D 44 4F 53 20 20-43 4F 4D 27 00 00 00 00   IBMDOS  COM'....
396F:0130  00 00 00 00 00 00 00 60-68 06 07 00 00 43 00 00   .......'h....C..
396F:0140  43 4F 4D 4D 41 4E 44 20-43 4F 4D 20 00 00 00 00   COMMAND COM ....
396F:0150  00 00 00 00 00 00 00 60-68 06 18 00 00 45 00 00   .......'h....E..
396F:0160  41 53 53 45 4D 42 4C 45-52 20 20 08 00 00 00 00   ASSEMBLER  .....
396F:0170  00 00 00 00 00 00 33 9C-B0 06 00 00 00 00 00 00   ......3.0.......
```

```
-D
396F:0180  46 57 20 20 20 20 20 20-43 4F 4D 20 00 00 00 00   FW      COM ....
396F:0190  00 00 00 00 00 00 00 00-6F 05 2A 00 80 AF 00 00   ........o.*../..
396F:01A0  46 57 20 20 20 20 20 20-4F 56 4C 20 00 00 00 00   FW      OVL ....
396F:01B0  00 00 00 00 00 00 00 00-72 05 56 00 81 02 00 00   ........r.V.....
396F:01C0  46 57 20 20 20 20 20 20-53 57 50 20 00 00 00 00   FW      SWP ....
396F:01D0  00 00 00 00 00 00 9B 8A-FF 06 57 00 00 C8 00 00   ..........W..H..
396F:01E0  43 4F 4E 46 49 47 20 20-44 41 54 20 00 00 00 00   CONFIG  DAT ....
396F:01F0  00 00 00 00 00 00 1D 82-A1 06 89 00 00 28 00 00   ........!....(..
```

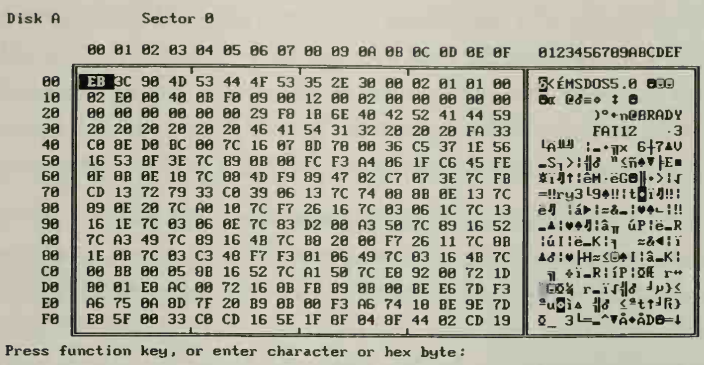
We will use a format much like this for Dskpatch, but with improvements. Dskpatch will be the equivalent of a full-screen editor for disk sectors. You will be able to display sectors on the screen and move the cursor about the sector display, changing numbers or characters as you want. You will also be able to write this altered sector back to the disk. That is why it is called Disk Patch—actually Dskpatch, because you cannot have more than eight characters in the name.

Dskpatch is the motivation for the procedures you will write. It is by no means an end in itself. In using Dskpatch as an example for this book, we will also manage to present many procedures that you will find useful when you attempt to write your own programs. That means you will find many general-purpose procedures for display output, display manipulation, keyboard input, and more.

Let's take a closer look at some of the improvements you will make to Debug's sector dump. The display from Debug only shows the "printable" characters—96 out of the 256 different characters that an IBM PC can display. Why is that? This occurs because MS-DOS was designed to run on many different computers, including computers that used simple computer terminals. The author of Debug chose to show a period for all other characters. Most computers are PC-compatibles, so there is no reason for Debug to continue to restrict its display, but no one at Microsoft ever changed the code.

Dskpatch is for PC compatibles, so you can display all 256 different characters with a bit of work. Using the DOS function 2 for character output, You can display almost all characters. However, DOS gives special meaning to some, such as 7, which rings the bell. There are characters for special codes like 7, and in Part III you will learn how to display them.

We will also make heavy use of the function keys so that you can display the next sector by pressing the F4 key. You will also be able to change any byte by moving the cursor to that byte and typing in a new number. It will be just like using a word processor, where you can easily change characters. More of these details will appear as you slowly build Dskpatch. Figure 12-2 shows what its normal display will look like—a vast improvement over the display from Debug.



```
Disk A          Sector 0

         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF

   00    EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00   δK ÉMSDOS5.0 ◘☺☺
   10    02 E0 00 40 0B F0 09 00 12 00 02 00 00 00 00 00   ☻α @♂≡○ ↕ ☻
   20    00 00 00 00 00 00 29 F8 1B 6E 40 42 52 41 44 59         )°←n@BRADY
   30    20 20 20 20 20 20 46 41 54 31 32 20 20 20 FA 33         FAT12   ·3
   40    C0 8E D0 BC 00 7C 16 07 BD 70 00 36 C5 37 1E 56   └Ä╨╝ |▬•╜p 6┼7▲V
   50    16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE   ▬S┐>|╣♂ ⁿ≤ñ♠▼╞E■
   60    0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB   ☼ï♫↑|êM·ëG☻╟•>|√
   70    CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C   ═‼ry3└9♠‼|t◘ï♫‼|
   80    89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13   ë♫ |á►|≈&▬|♥♠∟|‼
   90    16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52   ▬▲|♥♠♫|â╥ úP|ë▬R
   A0    7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B   |úI|ë▬K|╕  ≈&◄|ï
   B0    1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 03 16 4B 7C   ▲♂|♥├H≈≤☺♠I|♥▬K|
   C0    00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D    ╗ ♣ï▬R|íP|ΦÆ r↔
   D0    B8 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3   ╕☺Φ¼ r▬ï√╣♂ ╛µ}≤
   E0    A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D   ªu◙ì⌂ ╣♂ ≤ªt↑╛₧}
   F0    E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19   Φ_ 3└═▬^▼Å♦ÅD☻═↓

Press function key, or enter character or hex byte:
```

Figure 12-2: Example of Dskpatch's Display.

The Game Plan for Building Dskpatch

In Chapter 13, you will learn how to break your program into many different source files. Then, you will begin serious work on Dskpatch in Chapter 14. At the end, you will have nine source files for Dskpatch that have to be linked together. Even if you don't enter and run all these programs now, they'll be here when you're ready for them, or when you want to borrow some of the general-purpose procedures. In any case, you will get a better idea of how to write long programs as you read through the following chapters.

You have already created several useful procedures such as WRITE_HEX to write a byte as a two-digit hex number and WRITE_DECIMAL to write a number in decimal. Now, you will write some programs to display a block of memory in much the same way Debug's D command does. Start by displaying 16 bytes of memory, one line of Debug's display, and then work toward displaying 16 lines of 16 bytes each (half a sector). A full sector won't fit on the display at one time with the format we have chosen, so Dskpatch includes procedures for scrolling through a sector using the ROM BIOS—not DOS—interrupts. That will come much later, though, after you have built a full-screen display of half a sector.

Once you can dump 256 bytes from memory, you will build another procedure to read a sector from the disk into our area of memory. You will dump half a sector on the screen, and you will be able to use Debug to alter the program, so you can dump different sectors. At that point, you will have a functional, but not very attractive, display, so making it pretty comes next.

With a bit more work and some more procedures, you will rebuild the half-sector display to be much more pleasing aesthetically. It still won't be a full-screen display, so it will just scroll past like Debug's dump did. The full-screen display will come next, and you will learn about the ROM BIOS routines that allow you to control the display, move the cursor, etc. Then, you will be ready to learn how to use more ROM BIOS routines to print all 256 different characters.

Next will come the keyboard input and command procedures so you can start interacting with Dskpatch. About that time you will also need another course correction.

Summary

You have seen enough of the future here. You should have a better idea of where we're headed, so let's move on to the next chapter. Chapter 13 will lay the groundwork for modular design and teach you how to split a program into different source files. Then, in Chapter 14, you will write test procedures to display sections of memory.

C H A P T E R

Modular Design—Building Programs in Pieces

In this chapter you will learn how to break your programs into more than one file. You will also learn more about modular design, which is the foundation of well-organized programs. Finally, you will take a look at Microsoft's Programmers Workbench, which is a nice environment for building complex programs with many source files.

Files altered: VIDEO_IO.ASM and TEST.ASM

Disk files: VIDEO_13.ASM and TEST13.ASM

13

Topics Covered

Separate Assembling

The Three Laws of Modular Design

Using the Programmer's Workbench

Summary

Without modular design, Dskpatch wouldn't have been much fun to write. Using a modular design greatly eases the task of writing any but the smallest program. This chapter sets some ground rules for modular design which will be followed throughout the rest of the book. Let's begin by learning how to separate a large program into many different source files.

Separate Assembling

In Chapter 10, we added the procedure WRITE_DECIMAL to VIDEO_IO.ASM, and we also added a short test procedure called TEST_WRITE_DECIMAL. Let's take this test procedure out of VIDEO_IO.ASM and put it in a file of its own, called TEST.ASM. Then, you will assemble these two files separately and link them together into one program as shown in Listing 13-1. The TEST.ASM file is as follows:

Listing 13-1 The File TEST.ASM (TEST13.ASM)

```
DOSSEG
.MODEL  SMALL

.STACK

.CODE
        EXTRN   WRITE_DECIMAL:PROC

TEST_WRITE_DECIMAL      PROC
        MOV     DX,12345
        CALL    WRITE_DECIMAL
        MOV     AH,4Ch                     ;Return to DOS
        INT     21h
TEST_WRITE_DECIMAL      ENDP

        END     TEST_WRITE_DECIMAL
```

You have seen most of this source file before, but the EXTRN directive is new. The statement EXTRN WRITE_DECIMAL:PROC tells the assembler two things—that WRITE_DECIMAL is in another, *external*, file, and that it is a procedure. What kind of procedure (NEAR or FAR) depends on the MODEL directive. Since we have used .MODEL SMALL, which defines procedures to be NEAR, WRITE_DECIMAL is in the same segment. The assembler thus generates a NEAR CALL for this procedure. It would generate a FAR CALL if we had placed a FAR after WRITE_DECIMAL. You can use NEAR or FAR in place of the PROC in the EXTRN statement if you wanted to explicitly define the type of procedure, but it is better to let the .MODEL directive define the procedure types.

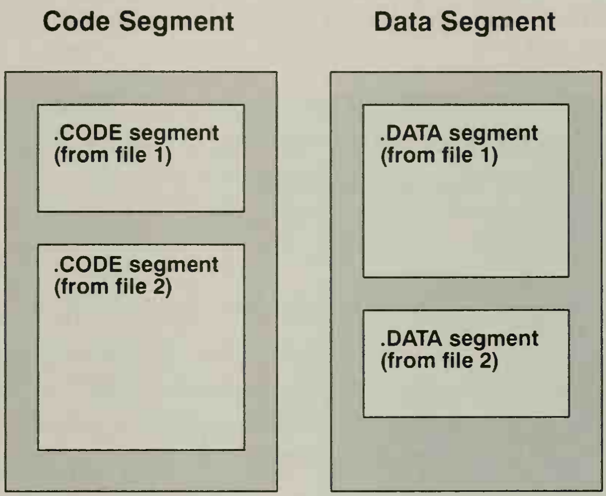
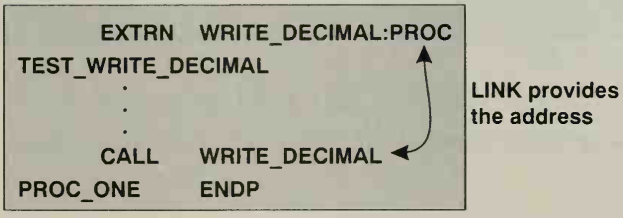

Code Segment
Data Segment
.CODE segment (from file 1)
.CODE segment (from file 2)
.DATA segment (from file 1)
.DATA segment (from file 2)

Figure 13-1: LINK stitches together segments from different files.



```
        EXTRN   WRITE_DECIMAL:PROC
TEST_WRITE_DECIMAL
          .
          .
          .
        CALL    WRITE_DECIMAL
PROC_ONE        ENDP
```

Figure 13-2: LINK assigns the addresses for external names.

These are about the only changes you need for separate source files until you begin to store data in memory. At that point, you will introduce another segment for data. Now, let's modify VIDEO_IO.ASM, and then assemble and link the two files. Remove the procedure TEST_WRITE_DECIMAL from VIDEO_IO.ASM because you have placed it in TEST.ASM and you do not need it in Video_io.

Finally, change END TEST_WRITE_DECIMAL at the end of VIDEO_IO.ASM to just END. Once again you moved the main procedure was moved to TEST.ASM. The procedures in VIDEO_IO.ASM are now *external* procedures. That is, they have no function by themselves; they must be linked to procedures that call them from other files. You don't need a name after the END directive in VIDEO_IO.ASM because your main program is now in TEST.ASM.

When you have finished making these changes, your VIDEO_IO.ASM source file should look something like the following listing. The complete listing is included in VIDEO_13.ASM on the disk.

```
.MODEL  SMALL
.CODE

        PUBLIC  WRITE_HEX
            .
            .
            .
WRITE_HEX       ENDP


        PUBLIC  WRITE_HEX_DIGIT
            .
            .
            .
WRITE_HEX_DIGIT ENDP


        PUBLIC  WRITE_CHAR
            .
            .
            .
WRITE_CHAR      ENDP
```

```
        PUBLIC  WRITE_DECIMAL
                .
                .
                .
WRITE_DECIMAL   ENDP


        END
```

Assemble these two files just as you assembled Video_io before, using ML /c followed by the file name. TEST.ASM knows all it needs to know about VIDEO_IO.ASM through the EXTRN statement. The rest will come when you link the two files.

Note

ML can assemble more than one file at a time. All you have to do is include all the files you want to assemble after the ML /c. For example, you can assemble both files being used here with this command:

```
ML /C TEST.ASM VIDEO_IO.ASM
```

You can create the TEST.EXE file directly without creating any of the OBJ files along the way. If you type ML without the /C switch, this tells ML to create an EXE file directly. To see how this works, delete the OBJ files from your disk, then type the following to create TEST.EXE:

```
ML TEST.ASM VIDEO_IO.ASM
```

When you assemble a program like this, the final program will always have the same name, but with an EXE extension, as the first file name in the list.

Now you should have the files TEST.OBJ and VIDEO_IO.OBJ. Use the following command to link these two files into one program named TEST.EXE:

```
C>LINK TEST VIDEO_IO;
```

LINK stitches the procedures of these two files together to create one file containing the entire program. It uses the first file name you entered as the name for the resulting EXE file, so you now have TEST.EXE.

That's it; you created one program from two source files. The final EXE program is identical in function to the COM version you created in Chapter 10 from the single file VIDEO_IO.ASM, when it contained the main procedure TEST_WRITE_DECIMAL.

We will make heavy use of separate source files from here on. Their value will become clearer as the procedures stack up. In the next chapter, you will write a test program to dump sections of memory in hex. You will usually write a simple test version of a procedure before writing the complete version. Doing so will allow you to see how to write a good final version, as well as to save effort and mental turmoil in the process. There are several other useful ways to save effort. They are called *The Three Laws of Modular Design.*

The Three Laws of Modular Design

These laws are summarized in Table 13-1. They aren't really *laws,* they are suggestions. But we will use them throughout this book. Define your own laws if you like; either way, stick to the same ones all the time. Your job will be much easier if you are consistent.

Table 13-1 The Three Laws of Modular Design

1. Save and restore all registers, unless the procedure returns a value in that register.
2. Be consistent about which registers you use to pass information. For example:

DL, DX	Send byte and word values
AL, AX	Return byte and word values
BX:AX	Return double-word values
DS:DX	Send and return addresses

CX	Repeat counts and other counts
CF	Set when there is an error; an error code should be returned in one of the registers, such as AL or AX.

3. Define all external interactions in the comment header:
   - Information needed on entry
   - Information returned (registers changed)
   - Procedures called
   - Variables used (read, written, and so on)

There is an obvious parallel between modular design in programming and modular design in engineering. An electrical engineer, for example, can build a very complicated piece of equipment from boxes that perform different functions, without knowing how each box works inside. But if each box uses different voltages and different connections, the lack of consistency creates a major headache for the engineer who must somehow provide a different voltage for each box and create special connections between boxes. Fortunately for the engineer, there are standards providing for only a small number of standard voltages. So, perhaps only four different voltages need to be provided instead of a different voltage for each box.

Modular design and standard interfaces are just as important in assembly-language programs, and that is why we will lay down the laws (so to speak), and use those laws from here on. As you will see by the end of this book, these rules will make your task much simpler. Let's take a look at these laws in detail.

Save and restore *all* registers, *unless* the procedure returns a value in that register.

There aren't that many registers in the 80x86. By saving registers at the start of a procedure, you free them for use within that procedure. But you must be careful to restore them at the end of the procedure. You will see us doing this in all of the procedures, with PUSH instructions appearing first in each procedure and POPs at the end.

The only exception is for procedures that must return some information to the calling procedure. For example, a procedure that reads a character from the keyboard must somehow return the character. Don't save any registers that are used to return information.

Short procedures also help the register-shortage problem. At times, you will write a procedure that is used by only one other procedure. Not only does this help with the shortage of registers, it also makes the program easier to write and read. You will see more of this as we write procedures for Dskpatch.

Be consistent about which registers you use to pass information.

Your job becomes simpler if standards are set for exchanging information between procedures. Use one register for sending information, and one for receiving information. You will also need to send addresses for long pieces of data. For this you will use the DS:DX pair of registers so that your data can be anywhere in memory. You will learn more about this when we introduce a new segment for data and begin to make use of the DS register.

The CX register is reserved for repeat counts. Soon you will write a procedure to write one character several times so that you can write 10 spaces by calling this procedure (WRITE_CHAR_N_TIMES) with CX set to 10. Use the CX register whenever you have a repeat count or when you want to return some count, such as the number of characters read from the keyboard (you will do this when we write a procedure named READ_STRING).

Finally, set the Carry Flag (CF) whenever there is an error, and clear it whenever there isn't an error. Not all procedures use carry flags. For example, WRITE_CHAR always works, so there is no reason to return an error report. But a procedure that writes to the disk can encounter many errors (no disk, write-protection, and so on). In this case, you will use a register to return an error code. There is no standard here because DOS uses different registers for different functions.

Define *all* external interactions in the comment header.

There is no need to learn how a procedure works if all you want to do is use it. This is why we place a detailed comment header before each procedure. This header contains *all* the information you need to know. It tells you what to place in each register before calling the procedure, and what information the procedure returns. Most procedures use registers for their variables, but some of the procedures use variables in memory. The comment header should say which of these memory variables are read and which are changed. Finally, each header should list other procedures called. An example of a full-blown header with much of this information is as follows:

```
;-----------------------------------------------------------------;
; This is an example of a full-blown header. This part would normally   ;
; be a brief description of what this procedure does. For example,      ;
; this procedure will write the message "Sector   " on the first line.  ;
;                                                                        ;
; On entry:     DS:DX   Address of the message "Sector   "              ;
; Returns:      AX      Error code if there was an error                 ;
;                                                                        ;
; Calls:        GOTO_XY, WRITE_STRING   (procedures called)             ;
; Reads:        STATUS_LINE_NO          (memory variables read only)    ;
; Writes:       DUMMY                   (memory variables altered)      ;
;-----------------------------------------------------------------;
```

Whenever you want to use any procedure you have written, you can just glance at this comment header to learn how to use it. There will be no need to delve into the inner workings of the procedure to find out what it does.

You may discover from time to time that your comment headers don't tell you enough so you can use the subroutine, and you have to actually read the code. When you find yourself doing this, consider rewriting your comment header so it does a better job of explaining how the subroutine works. If you don't rewrite the header, you may find yourself again reading the code.

These laws make assembly language programming easier, and we will be certain to abide by them, but often not on the first try. The first version of a procedure or program is a test case. Frequently, we don't know exactly how to write the program we have in mind, so on these "rough drafts," we will write the program without concern for the laws of modular design. We will just plow through and get something that works. Then we can backtrack and do a good job by rewriting each procedure to conform to these laws.

Programming is a process that goes by leaps and bounds. Throughout this book we will show much, but not all, of the stuttering that went into writing

Dskpatch. There is not room enough to contain all the versions we wrote before we settled on the final version. Our first tries often bear very little resemblance to the final versions you will see. When you write programs, don't worry about getting everything right the first time. Be prepared to rewrite each procedure as you learn more about what you really want.

In the next chapter, you will build a simple test program to print a block of memory. It won't be the final version; we will go through others before we are satisfied, and even then, there will be other changes we would like to make. The moral is: A program is never done . . . but it must be stopped somewhere.

Using the Programmer's Workbench

The Microsoft MASM 6.0 package includes a full programming environment, called the *Programmer's Workbench.* This can make your job of building assembly language programs much easier. In reality, some programmers love the Programmer's Workbench (PWB), and others hate it. Since there are a number of people who don't like PWB, we won't force you to use it in this book; we will continue to use ML and LINK directly. For those of you who feel adventuresome, PWB has a lot to offer.

In this section we would like to give you an introduction to using PWB. We will use PWB to build TEST.EXE so you can see how you might use PWB to build your own assembly-language programs that use multiple files. We will assume that you asked to install PWB when you installed MASM 6—if you didn't, go back and do so now.

When you first start PWB you will see a screen like the one shown in Figure 13-3. At this point you could simply start typing a program, then use the pull-down menus to save the file. But since you already have two files, you will take a different approach.

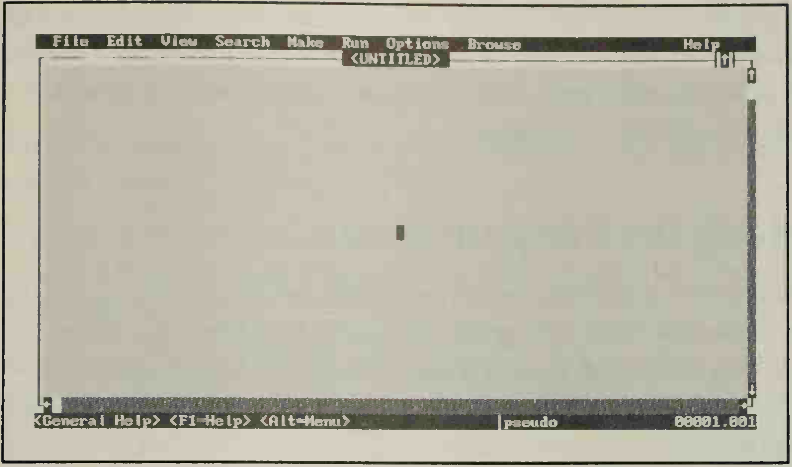

File Edit View Search Make Run Options Browse Help
<UNTITLED>
<General Help> <F1=Help> <Alt=Menu>
pseudo
00001.001

Figure 13-3: Programmer's Workbench when you first start it.

Using PWB's Pull-Down Menus

You are going to create a *program list*, which is a list of files that constitute a program. PWB can use this list of files to automatically create the EXE file, as you will see. Before we go on, a few words about pull-down menus and dialog boxes. PWB makes heavy use of both features, so you will need to know how to use them. To pull down a menu, you have two choices, depending on whether you want to use the keyboard or the mouse.

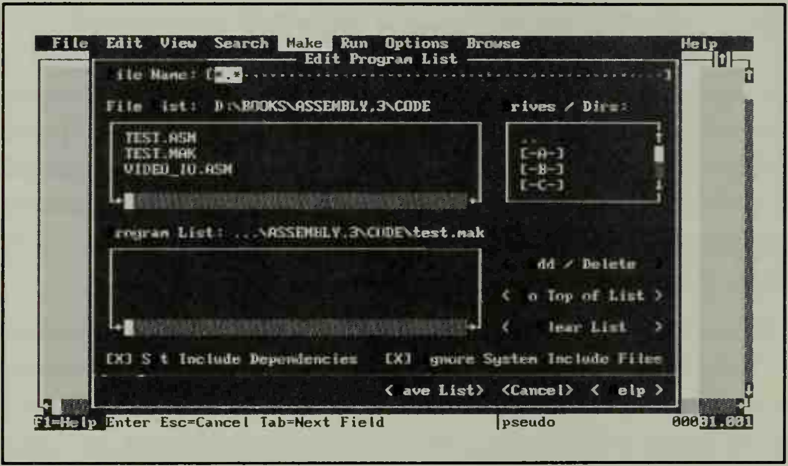
The keyboard interface centers around the Alt key. When you press the Alt key, PWB highlights the first letter of each menu item, which indicates that you can press Alt+*letter*. For example, you will want to pull down the Make menu, by pressing Alt+M. Once the menu is down, either use the cursor keys and press Enter to select an item from the list, or type one of the highlighted letters you see in the pull-down menu. In either case, you will notice that only one menu item, `Set Program List...` has a letter highlighted. This happens because all the other menu items are grayed out, which means their options are not available.

Using pull-down menus with the mouse is even easier. Simply click the left button on the menu bar at the top of the screen on any of the menu titles, such as Make, and a menu will drop down. You can then click on any of the menu items to select that item.

Setting the Program List

Now pull down the Make menu and select the item `Select Program List...` You will see the "Select Program List" dialog box, which is a dialog box that allows you to open an existing Program List. You need to create a new program list, which you can do by typing a name and pressing Enter. Type TEST and press Enter.

PWB will display another dialog box asking if you want to create the file "test.mak." Press Enter or click on the `<Yes>` button—you do want to create a new program list. Now you will now see the dialog box in Figure 13-4.



```
 File  Edit  View  Search  Make  Run  Options  Browse                    Help
                        Edit Program List
  ile Name: [*.*.............................................]
 File  ist:  D:\BOOKS\ASSEMBLY.3\CODE                 rives / Dirs:
  TEST.ASM                                             ..
  TEST.MAK                                             [-A-]
  VIDEO_IO.ASM                                         [-B-]
                                                       [-C-]
 rogram List:  ...\ASSEMBLY.3\CODE\test.mak
                                                       dd / Delete
                                                    <  o Top of List >
                                                    <   lear List    >
 [X] S t Include Dependencies    [X]  gnore System Include Files
                                   < ave List>  <Cancel>  <  elp >
F1=Help Enter Esc=Cancel Tab=Next Field        |pseudo              00001.001
```

Figure 13-4: Use the Edit Program List dialog box to create and change program lists.

This dialog box is where you actually build program lists, or modify them once you have already built a list. You want to add two files to this program list:

TEST.ASM and VIDEO_IO.ASM, both of which should be visible in the File List window near the top of this dialog box. The steps you can follow to add these two files to the Program List are as follows:

1. Press Tab until the cursor moves to the File List window (probably a single Tab).
2. Use the up and down cursor keys to highlight TEST.ASM, then press the Enter key. You should now see TEST.ASM in the Program List window.
3. Repeat step 2 to add VIDEO_IO.ASM to the Program List.
4. Press Alt+S or click on the `<Save List>` button to save this program list.

PWB will think for a couple of seconds, then you will see the mostly-blank screen that you saw when you first started PWB. It may not seem like you have done much, but you have.

Building TEST.EXE

Pull down the Make menu again and you will notice it is quite different. Now all the items in this menu are enabled. But you will also notice that `test.exe` appears in the Build menu item. This means you can build TEST.EXE simply by pulling down the Make menu and selecting Build. What is even better is that PWB keeps track of which files you have changed, and it only assembles the files that need to be changed.

You can tell which Program List is currently selected by looking at the `Edit Program List...` *item in the Make menu. If you see a name, such as* `test` *after this item, you know that the Program List with that name is currently selected.*

If you try building TEST.EXE, you will get the following error message:

```
LINK : warning L4050: file not suitable for /EXEPACK; relink without
```

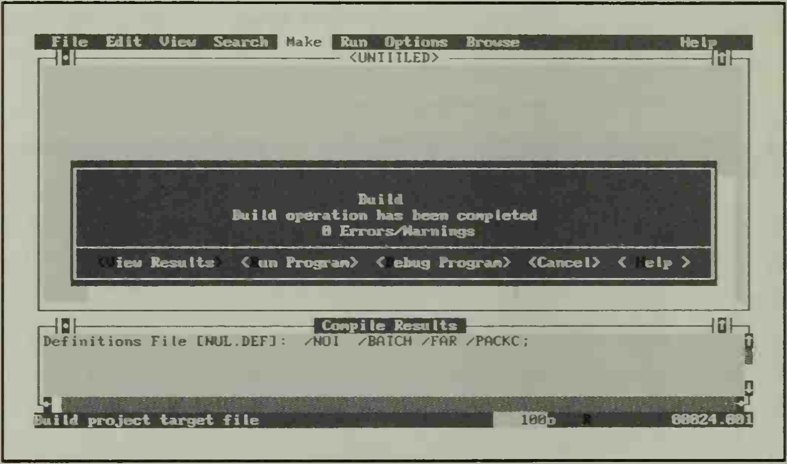
What does this mean? By default, PWB tells the linker to try to *pack* your EXE file. Packed EXE files are files that have been compressed so they take up less disk space. The problem here is that your program is so small it cannot be packed without actually growing in size. The file grows because LINK has to add a small program to any EXE file it packs to unpack that program when its loaded into memory. Since this unpacker is larger than the entire TEST.EXE program, the LINK failed.

What you have to do is change the way PWB tells LINK to create your EXE file. Pull down the Options menu and select `LINK Options...`. Then in the LINK Options dialog box press Alt+R or click on `<Set Release Options...>` to bring up yet another dialog box. Press Alt+E or click on `Pack EXE file` to make sure this option is not checked (it is checked whenever there is an `x` in square brackets, such as `[X]`). You want to make sure it reads the following:

```
[ ] Pack EXE file
```

Finally, click on the <OK> button in each of these dialog boxes (or use the Tab key to move the cursor to the <OK> button, then press Enter). Now select Build from the Make menu. It should work.

To reward you for all your hard work, you will see the final dialog box, like the one in Figure 13-5. You can then elect to run your program.


File Edit View Search Make Run Options Browse Help
<UNTITLED>
<View Results> <Run Program> <Debug Program> <Cancel> < Help >
Compile Results

```
Build
Build operation has been completed
0 Errors/Warnings
```

```
Definitions File [NUL.DEF]:  /NOI  /BATCH /FAR /PACKC;
```

```
Build project target file          100b  R          00024.001
```

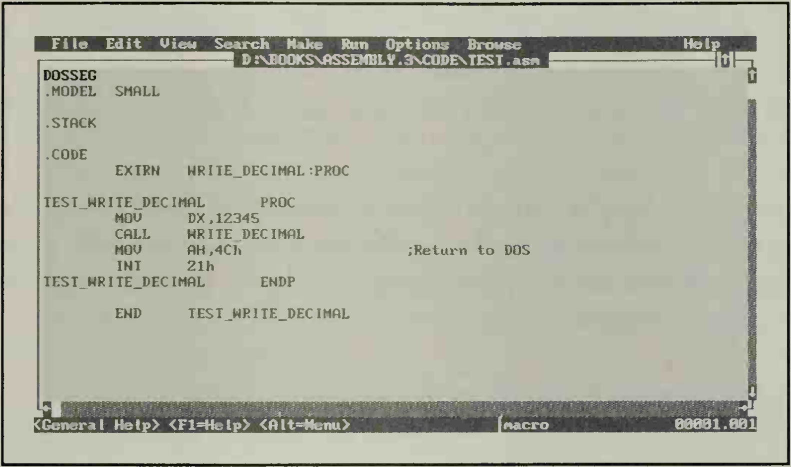
Figure 13-5: Final dialog box when PWB finished building your program.

Editing Files in PWB

If all PWB could do is build programs, it would not be that useful. Fortunately, PWB can do much more for you. For starters, you can edit any of the files in your Program List. An easier way to get to your files, however, is another feature called Browse, which enables you to jump very quickly to any place in your program.

Before you can use the Browse feature, however, you need to enable it. Pull down the Options menu and select the "Browse Options..." item. Once the dialog box appears, press Alt+B to check the "Generate Browse Information" option (so it has [X] next to it) and press Enter. Build your program again by selecting Build from the Make menu. Press Esc when you see the final dialog box saying that the Build operation finished. Whenever you rebuild your program with the Generate Browse Information option checked, PWB keeps a lot of information on your program which it needs for all the items in the Browse menu.

Pull down the Browse menu and select `View Relationship`. You will see a dialog box that lists two files in it. These are the two files that make up your Program List. Select the file TEST.ASM and press Alt+G (Goto). You will now have the file TEST.ASM visible in an edit window (see Figure 13-6) and you can edit TEST.ASM.

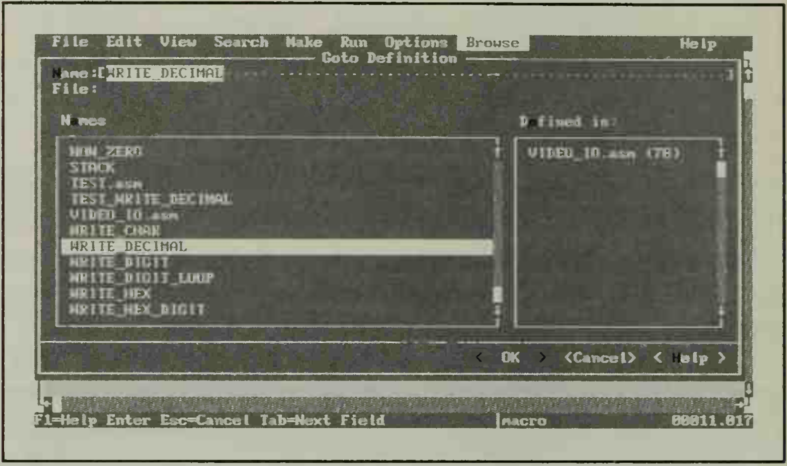

File Edit View Search Make Run Options Browse Help
D:\BOOKS\ASSEMBLY.3\CODE\TEST.asm
<General Help> <F1=Help> <Alt=Menu>
macro
00001.001

```
DOSSEG
.MODEL  SMALL

.STACK

.CODE
        EXTRN   WRITE_DECIMAL:PROC

TEST_WRITE_DECIMAL      PROC
        MOV     DX,12345
        CALL    WRITE_DECIMAL
        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
TEST_WRITE_DECIMAL      ENDP

        END     TEST_WRITE_DECIMAL
```

Figure 13-6: TEST.ASM in its own edit window.

Jump to Definitions

Move the cursor to the name WRITE_DECIMAL after the CALL. Now pull down the Browse menu and select "Goto Definition." You will see the dialog box in Figure 13-7, with WRITE_DECIMAL highlighted in the list. This list shows all of the procedures, labels, etc. in your program. Press Enter, and now you are looking at the definition of WRITE_DECIMAL in the file VIDEO_IO.ASM.


File Edit View Search Make Run Options Browse Help
Goto Definition
< OK > <Cancel> < Help >
F1=Help Enter Esc=Cancel Tab=Next Field
macro
00011.017

```
Name:[WRITE_DECIMAL
File:

Names                                         Defined in:
NON_ZERO                                      VIDEO_IO.asm (78)
STACK
TEST.asm
TEST_WRITE_DECIMAL
VIDEO_IO.asm
WRITE_CHAR
WRITE_DECIMAL
WRITE_DIGIT
WRITE_DIGIT_LOOP
WRITE_HEX
WRITE_HEX_DIGIT
```

Figure 13-7: This dialog box enables you to jump quickly to any part of your program.

You can easily jump to any definition in any of the files of your project by using this menu item. You can also find all of the places where a particular subroutine is used in other parts of your program by using the Goto Definition... item in the Browse menu.

Note

The information used by the Browse menu comes from the last build you did on your program. It won't reflect any changes you have made to your files since the last time you built your program from *within* PWB. Changes you make outside PWB won't show up inside PWB until you use PWB's Build command again.

Loading the Program List

PWB has one rather rude habit that probably keeps a lot of programmers from using it. When you exit and then start PWB again, it remembers all the files you had opened, but it *does not* remember the last Program List that you were using. You have to explicitly load the Program List each time you start PWB. You do this by using the Set Program List... from the Make menu to select the .MAK file you want to use. Fortunately, there is an easier solution.

There is a little-known switch you can use to have PWB automatically load the last Program list you were using. You can also use the /PL switch (this *must* be in uppercase letters). Type the following:

```
C>pwb /PL
```

There are far more features of PWB than we can show here, but this should give you an idea of how to use PWB. You might want to try using PWB for all the programs in the rest of this book. You can also use Microsoft's wonderful CodeView debugger from within PWB.

For the rest of this book, however, we will mainly show you the details of using ML and LINK to do all the work. There are two reasons for this. First, this will give you a better understanding of all the steps and other tools out there; second, there are many people who detest PWB, and they would not be happy if we used PWB for the rest of this book. To back up that claim, most programmers we know like to use the Brief editor, along with the Make program you will learn in Chapter 15. I personally don't know of any programmers who use PWB.

Summary

This has been a chapter for you to remember and use in the future. We began by teaching you how to separate a program into a number of different source files that can be assembled independently, then stitched together with the linker. We used the PUBLIC and EXTRN directives to inform the linker that there are connections between different source files. PUBLIC says that other source files can CALL the procedures named after PUBLICs, while EXTRN tells the assembler that the procedure you want to use is in another file.

Then we moved on to The Three Laws of Modular Design. These rules are meant to make your programming job simpler, so use them when writing your own programs (just as you will see us use them in this book). You will find it easier to write, debug, and read programs if they conform to these three laws.

Finally, we covered Microsoft's Programmers Workbench, which has a lot of nice features and is nice to work with once you get accustomed to its quirks. In all fairness, many programming systems have quirks, so it is a matter of what you are used to working with and what your co-workers use.

CHAPTER

Dumping Memory

In this chapter you will learn how to use the different addressing modes to access memory. You will also learn how to store data in the data segment and how to set DS so it points to this data segment. The changes you will make to your program will allow you to display an area of memory in hex and as characters, much as Debug can display an area of memory.

Files altered: DISP_SEC.ASM, CURSOR.ASM, and VIDEO_IO.ASM

Disk files: DISP_S14.ASM, CURSOR14.ASM, and VIDEO_14.ASM

14

Topics Covered

Using Addressing Modes to Access Memory

Using the Data Segment

Base-Relative Addressing

Setting DS to Point to your Data Segment

Adding Characters to the Dump

Dumping 256 Bytes of Memory

Summary

From here on, we will concentrate on building Dskpatch in much the same way we originally wrote it. Some of the instructions in new procedures may be unfamiliar. We will explain each briefly as we come across them. (For detailed information, you will need a book that covers all of the instructions in detail. Most reference books that cover any of the 80x86 microprocessors have all the information you should need.)

Rather than cover all the 80x86 instructions, we will concentrate on new concepts, such as the different modes of addressing memory, which we will cover in this chapter. In Part III, we will move even farther away from the details of instructions and begin to see information specific to writing DOS programs that use the screen and keyboard.

Now, you will learn about *addressing modes* by writing a short test program to dump 16 bytes of memory in hex notation. To begin, you need to learn how to use memory as variables.

Using Addressing Modes to Access Memory

You have seen two addressing modes; they are known as the *register* and *immediate* addressing modes. The first mode you learned about was the register mode, which uses registers as variables. For example, the following instruction uses the two registers AX and BX as variables.

```
MOV     AX,BX
```

Then, you moved on to the immediate addressing mode, in which you moved a number directly into a register, as in the following example:

```
MOV     AX,2
```

This example moves the byte or word of memory *immediately* following the instruction into a register. In this sense, the MOV instruction in our example is one byte long, with two more bytes for the data (0002):

```
396F:0100 B80200          MOV     AX,0002
```

The instruction is B8h, and the two bytes of data (02h and 00h) follow this (remember that the 80x86 stores the low byte, 02h, first in memory).

Now you will learn how to use memory as a variable. The immediate mode allows you to read the piece of fixed memory immediately following that one instruction, but it does not allow you to change memory. For this, you will need other addressing modes.

Let's begin with an example. The following program reads 16 bytes of memory, one byte at a time. Each byte is displayed in hex notation, with a single space between each of the 16 hex numbers. Enter the program into the file DISP_SEC.ASM and assemble it. Enter the Listing 14-1 into the new file DISP_SEC.ASM.

Listing 14-1 The New File DISP_SEC.ASM

```
DOSSEG
.MODEL  SMALL

.STACK

.DATA

        PUBLIC  SECTOR
SECTOR  DB      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h  ;Test pattern
        DB      18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh


.CODE

        EXTRN   WRITE_HEX:PROC
        EXTRN   WRITE_CHAR:PROC
;-----------------------------------------------------------------------;
; This is a simple test program to dump 16 bytes of memory as hex      ;
; numbers, all on one line.                                             ;
;-----------------------------------------------------------------------;
DISP_LINE       PROC
        MOV     AX,DGROUP                       ;Put data segment into AX
        MOV     DS,AX                           ;Set DS to point to data

        XOR     BX,BX                           ;Set BX to 0
        MOV     CX,16                           ;Dump 16 bytes
HEX_LOOP:
        MOV     DL,SECTOR[BX]                   ;Get 1 byte
        CALL    WRITE_HEX                       ;Dump this byte in hex
```

continues

Listing 14-1 continued

```
        MOV     DL,' '                      ;Write a space between numbers
        CALL    WRITE_CHAR
        INC     BX
        LOOP    HEX_LOOP

        MOV     AH,4Ch                      ;Return to DOS
        INT     21h
DISP_LINE       ENDP


        END     DISP_LINE
```

Try your new program to see how it works. Assemble Disp_sec.

You are ready to link DISP_SEC.OBJ and VIDEO_IO.OBJ and create an EXE file named DISP_SEC.EXE. LINK creates a program by putting the pieces together in the same order as the names on the command line. Since you want the main procedure to appear at the start of the program, the first file name in the LINK command needs to be the name of the file that contains the main procedure (Disp_sec in this case). A semicolon must appear at the end of the list of files, so type the following command:

```
C>LINK DISP_SEC VIDEO_IO;
```

Linking will always be the same, with more names before the semicolon when you have more files. The main procedure should always be in the first file listed. In general, the preceding step for the files *file1*, *file2*, and so on, is as follows:

```
LINK file1 file2 file3 ...;
```

Now, run the EXE file. If you don't see the following when you run the program, go back and check carefully for a mistake.

```
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
```

Now let's see how Disp_sec works. The following instruction uses a new addressing mode known as *Indirect Memory Addressing*—addressing memory through the *Base* register with *offset*, or more simply, *Base Relative*.

```
MOV     DL,SECTOR[BX]           ;Get 1 byte
```

In order to see what this really means, you need to first learn more about segments.

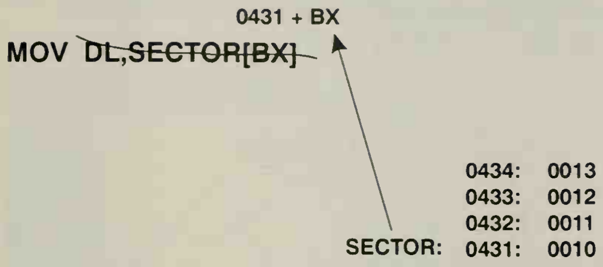


Figure 14-1: Translation of SECTOR[BX].

Using the Data Segment

Looking at Disp_sec, you will see that the label SECTOR appears after .DATA. The .DATA directive declares a data segment that is used for memory variables. (The name of the segment created by .DATA is _DATA.) Any time you want to store and read data in memory, you will set aside some space in this segment. We will get back to memory variables, but first a little more about segments.

The .MODEL SMALL directive creates what Microsoft calls a small memory-model program. Small programs are defined as programs that have up to 64K of code and up to 64K of data. In other words, one segment for code and one segment for data. Since both the data (defined by .DATA) and the stack (defined by .STACK) are data, they are put into a single segment as shown in Figure 14-2.

This *grouping* of the stack and data segments into one segment is handled by a mechanism in the assembler called *groups*. In particular, the assembler creates a group called DGROUP that creates a single segment out of all the segments used for data. So far you have learned the .DATA and .STACK directives. There are several other data directives that create segments in this group (you will see another later in this book). Fortunately, the .MODEL, .DATA, and .STACK directives handle all of this behind the scenes. Knowing some of what happens behind the scenes, however, will come in use later when you look at memory maps to see how programs are put together.

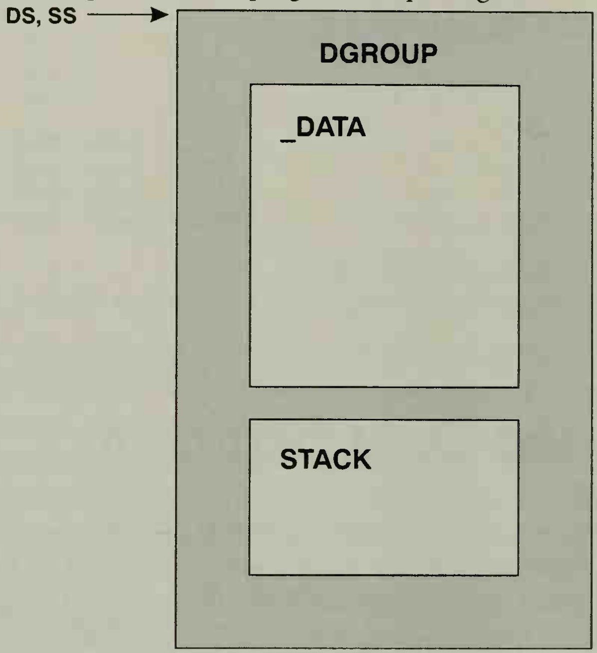


Figure 14-2: Stack and data in one segment group (DGROUP).

Another thing that happens automatically as a result of the DOSSEG directive is that the STACK segment is loaded into memory above the DATA segment. The data segment we created has data in it (10h, 11h, 12h, and so on) that needs to be in the EXE file so it can be copied into memory when the program is run by DOS. The stack, on the other hand, needs to take space in memory, but the stack's memory does not need to be initialized (only SS:SP has to be set). So by putting the stack segment after the data segment, you don't need to set aside space on the disk for the stack (see Figure 14-3).

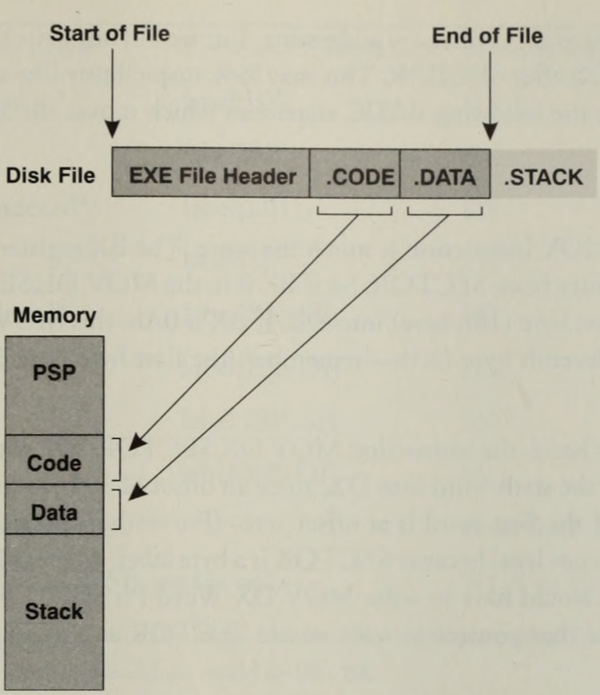


Figure 14-3: The stack segment uses no disk space.

Base-Relative Addressing

It is time to get back to the base-relative addressing mode. The following two lines set aside 16 bytes of memory in the data segment starting at SECTOR, which the assembler converts to an address.

```
SECTOR  DB      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h  ;Test pattern
        DB      18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh
```

DB, you may recall, stands for *Define Byte.* The numbers after each DB are initial values. When you first start DISP_SEC.EXE, the memory starting at SECTOR will contain 10h, 11h, 12h, and so on. If we wrote the command below the instruction would move the first byte (10h) into the DL register.

```
MOV     DL,SECTOR
```

This is known as *direct* memory addressing. But we did not write that. Instead, we placed [BX] after SECTOR. This may look suspiciously like an index into an array, like the following BASIC statement which moves the 10th element of L into K.

```
K = L(10)
```

In fact, the MOV instruction is much the same. The BX register contains an *offset* in memory from SECTOR. So if BX is 0, the MOV DL,SECTOR[BX] moves the first byte (10h here) into DL. If BX is 0Ah, this MOV instruction moves the eleventh byte (1Ah—remember, the first byte is at BX = 0) into DL.

On the other hand, the instruction MOV DX,SECTOR[BX] with BX = 0Ah would move the sixth word into DX, since an offset of 10 bytes is the same as 5 words, and the first word is at offset zero. (For enthusiasts this last MOV instruction is not legal because SECTOR is a byte label, whereas DX is a word register. You would have to write MOV DX,Word Ptr SECTOR[BX] to tell the assembler that you really want to use SECTOR as a word label in this instruction.)

There are many other addressing modes and some will be covered later. All of the addressing modes are summarized in Table 14-1.

Table 14-1 Addressing Modes

Addressing Mode	*Format of Address*	*Segment Register Used*
Register	register (such as AX)	None
Immediate	data (such as 12345)	None
	Memory Addressing Modes	
Register Indirect	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS

Addressing Mode	*Format of Address*	*Segment Register Used*
Base Relative*	label[BX]	DS
	label[BP]	SS
Direct Indexed*	label[DI]	DS
	label[SI]	DS
Base Indexed*	label[BX+SI]	DS
	label[BX+DI]	DS
	label [BP+SI]	SS
	label[BP+DI]	SS
String Commands:		Read from DS:SI
(MOVSW, LODSB, *and so on)*		Write to ES:DI

* Label[...] can be replaced by [disp+...], where *disp* is a displacement. Thus, we could write [10+BX] and the address would be 10 + BX.

Setting DS to Point to Your Data Segment

This discussion has glossed over one minor detail. In Chapter 11 we mentioned that both the DS and ES registers point to the PSP, not to your data segment when DOS starts the program. The first two lines in DISP_LINE set DS so it points to our data segment as follows:

```
MOV     AX,DGROUP                  ;Put data segment into AX
MOV     DS,AX                      ;Set DS to point to data
```

The first line moves the segment address for our data group (called DGROUP) that contains .DATA and .STACK into the AX register. The second line sets DS so it points to your data.

There is one sticky point here. If you remember the discussions about the segment registers, we said that the segment used for programs depends on how much memory is already in use. In other words, you cannot know the value of DGROUP until DOS loads the program in memory. How, then do you know what number to load into AX?

There is a small header at the start of each EXE file that contains a list of addresses in your program that have to be calculated. DOS uses this information to calculate the value of DGROUP and update the value in the MOV AX,DGROUP instruction when it loads DISP_SEC.EXE into memory. This process is known as relocation. You will see how DOS does relocation in Chapter 28.

There is another fine point of writing programs for the 80x86 family of microprocessor. Notice that we set the value of DS with two instructions rather than the single instruction, as follows.

```
MOV       DS,DGROUP
```

You need two instructions because you cannot move a number directly into a segment register on the 80x86; first you have to move the segment number into the AX register. Requiring two instructions, rather than one, simplified the design of the original 80x86 microprocessor. This made it less expensive to manufacture but more difficult to program.

Adding Characters to the Dump

You are almost finished writing the procedure that creates a dump display similar to Debug's. So far, you have dumped the hex numbers for one line; in the next step, you will add the character display following the hex display. The new version of DISP_LINE (in DISP_SEC.ASM), with a second loop added to display the characters, is as follows:

Listing 14-2 Changes to DISP_LINE in DISP_SEC.ASM

```
DISP_LINE       PROC
        MOV     AX,DGROUP                   ;Put data segment into AX
        MOV     DS,AX                       ;Set DS to point to data
        XOR     BX,BX                   ;Set BX to 0
        MOV     CX,16                   ;Dump 16 bytes
HEX_LOOP:
        MOV     DL,SECTOR[BX]           ;Get 1 byte
        CALL    WRITE_HEX               ;Dump this byte in hex
        MOV     DL,' '                  ;Write a space between numbers
        CALL    WRITE_CHAR
        INC     BX
        LOOP    HEX_LOOP

        MOV     DL,' '                  ;Add another space before characters
        CALL    WRITE_CHAR
        MOV     CX,16
        XOR     BX,BX                   ;Set BX back to 0
ASCII_LOOP:
        MOV     DL,SECTOR[BX]
        CALL    WRITE_CHAR
        INC     BX
        LOOP    ASCII_LOOP

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
DISP_LINE       ENDP
```

Assemble this, link it to Video_io, and try it. Figure 14-4 shows the output you should see.

```
C>disp_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ►◄↕‼¶§▬↨↑↓→←∟↔▲▼
C>_
```

Figure 14-4: DISP_LINE's output.

Try changing the data to include a 0Dh or a 0Ah. You will see a rather strange display, because 0Ah and 0Dh are the characters for the line-feed and carriage-return characters. DOS interprets these as commands to move the cursor, but we would like to see them as just ordinary characters for this part of the display. To do this, you must change WRITE_CHAR to print *all* characters, without applying any special meaning. You will do that in Part III, but for now, rewrite WRITE_CHAR slightly so that it prints a period in place of the low characters (between 0 and 1Fh), as you can see in Figure 14-5.

```
C>disp_sec
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F  ................
C>_
```

Figure 14-5: Modified version of DISP_LINE.

Replace the WRITE_CHAR in VIDEO_IO.ASM with the following new procedure in Listing 14-3:

Listing 14-3 A New WRITE_CHAR in VIDEO_IO.ASM (Complete listing in VIDEO_14.ASM)

```
        PUBLIC  WRITE_CHAR
;-----------------------------------------------------------------------;
; This procedure prints a character on the screen using the DOS         ;
; function call. WRITE_CHAR replaces the characters 0 through 1Fh with  ;
; a period.                                                             ;
;                                                                       ;
; On entry:     DL      byte to print on screen.                        ;
;-----------------------------------------------------------------------;
WRITE_CHAR      PROC
        PUSH    AX
        PUSH    DX
        CMP     DL,32                   ;Is character before a space?
        JAE     IS_PRINTABLE            ;No, then print as is
        MOV     DL,'.'                  ;Yes, replace with a period
IS_PRINTABLE:
        MOV     AH,2                    ;Call for character output
        INT     21h                     ;Output character in DL register
        POP     DX                      ;Restore old value in AX and DX
        POP     AX
        RET
WRITE_CHAR      ENDP
```

Try this new procedure with Disp_sec, and change the data to various characters to check the boundary conditions.

Dumping 256 Bytes of Memory

Now you have learned how to dump one line, or 16 bytes, of memory. The next step is to dump 256 bytes of memory. This happens to be exactly half the number of bytes in a sector, so you are working toward building a display of half a sector. You still have many more improvements to make; this is just a test version.

You will need two new procedures and a modified version of DISP_LINE. The new procedures are DISP_HALF_SECTOR, which will soon evolve into a finished procedure to display half a sector, and SEND_CRLF, which sends the cursor to the beginning of the next line (CRLF stands for *Carriage Return-Line Feed*, the pair of characters that move the cursor to the next line).

SEND_CRLF is very simple, so let's start with it. Place the following procedure in Listing 14-4 into a file called CURSOR.ASM.

Listing 14-4 The New File CURSOR.ASM (CURSOR14.ASM)

```
CR      EQU     13                      ;Carriage return
LF      EQU     10                      ;Line feed

.MODEL  SMALL
.CODE

        PUBLIC  SEND_CRLF
;-------------------------------------------------------------------;
; This routine just sends a carriage return-line feed pair to the   ;
; display, using the DOS routines so that scrolling will be handled ;
; correctly.                                                        ;
;-------------------------------------------------------------------;
SEND_CRLF       PROC
        PUSH    AX
        PUSH    DX
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
```

continues

Listing 14-4 continued

```
          POP      DX
          POP      AX
          RET
SEND_CRLF          ENDP


          END
```

This procedure sends a Carriage Return and Line Feed pair, using the DOS function 2 to send characters. The following statement uses the EQU directive to define the name CR to be equal to 13.

```
CR        EQU      13                          ;Carriage return
```

So the instruction MOV DL,CR is equivalent to MOV DL,13. As shown in Figure 14-6, the assembler substitutes 13 whenever it sees CR. Likewise, it substitutes 10 whenever it sees LF.

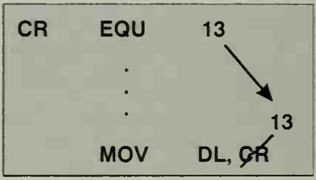


Figure 14-6: The EQU directive lets us use names in places of numbers.

Note

From here on, we will show the changes in our programs so you won't have to check each line to see if it's new or different. Additions to our programs will be shown against a gray background, and text you should delete will be printed with a line through the text:

Add or change lines displayed against a gray background

~~Delete text shown in strike-through characters~~

The file Disp_sec now needs much work. Here's the new version of DISP_SEC.ASM as shown in Listing 14-5.

Listing 14-5 The New Version of DISP_SEC.ASM (Complete listing in DISP_S14.ASM)

```
DOSSEG

.MODEL  SMALL

.STACK

.DATA
~~SECTOR  DB      10h, 11h, 12h, 13h, 14h, 15h, 16h, 17h  ;Test pattern~~
~~        DB      18h, 19h, 1Ah, 1Bh, 1Ch, 1Dh, 1Eh, 1Fh~~
SECTOR  DB      16 DUP (10h)
        DB      16 DUP (11h)
        DB      16 DUP (12h)
        DB      16 DUP (13h)
        DB      16 DUP (14h)
        DB      16 DUP (15h)
        DB      16 DUP (16h)
        DB      16 DUP (17h)
        DB      16 DUP (18h)
        DB      16 DUP (19h)
        DB      16 DUP (1Ah)
        DB      16 DUP (1Bh)
        DB      16 DUP (1Ch)
        DB      16 DUP (1Dh)
        DB      16 DUP (1Eh)
        DB      16 DUP (1Fh)

.CODE

        PUBLIC  DISP_HALF_SECTOR
        EXTRN   SEND_CRLF:PROC
;----------------------------------------------------------------------;
; This procedure displays half a sector (256 bytes)                    ;
;                                                                      ;
; Uses:         DISP_LINE, SEND_CRLF                                   ;
;----------------------------------------------------------------------;
DISP_HALF_SECTOR        PROC
        MOV     AX,DGROUP               ;Put data segment into AX
```

continues

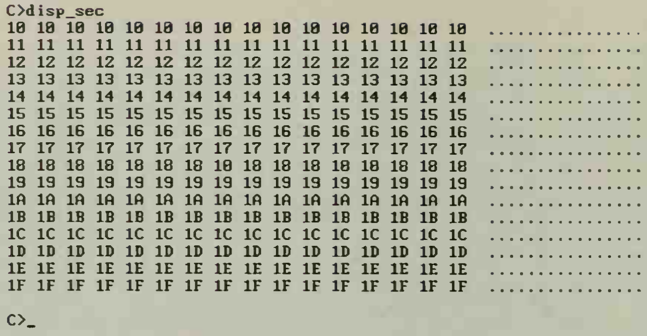
Listing 14-5 continued

```
        MOV     DS,AX                       ;Set DS to point to data

        XOR     DX,DX                       ;Start at beginning of SECTOR
        MOV     CX,16                       ;Display 16 lines
HALF_SECTOR:
        CALL    DISP_LINE
        CALL    SEND_CRLF
        ADD     DX,16
        LOOP    HALF_SECTOR

        MOV     AH,4Ch                      ;Return to DOS
        INT     21h
DISP_HALF_SECTOR        ENDP


        PUBLIC  DISP_LINE
        EXTRN   WRITE_HEX:PROC
        EXTRN   WRITE_CHAR:PROC
;-----------------------------------------------------------------------;
; This procedure displays one line of data, or 16 bytes, first in hex,  ;
; then in ASCII.                                                        ;
;                                                                       ;
; On entry:     DS:DX   Offset into sector, in bytes.                   ;
;                                                                       ;
; Uses:         WRITE_CHAR, WRITE_HEX                                   ;
; Reads:        SECTOR                                                  ;
;-----------------------------------------------------------------------;
DISP_LINE       PROC
~~        MOV     AX,DGROUP                   ;Put data segment into AX~~
~~        MOV     DS,AX                       ;Set DS to point to data~~

~~        XOR     BX,BX~~
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                       ;Offset is more useful in BX
        MOV     CX,16                       ;Dump 16 bytes
        PUSH    BX                          ;Save the offset for ASCII_LOOP
HEX_LOOP:
        MOV     DL,SECTOR[BX]               ;Get 1 byte
        CALL    WRITE_HEX                   ;Dump this byte in hex
        MOV     DL,' '                      ;Write a space between numbers
        CALL    WRITE_CHAR
        INC     BX
        LOOP    HEX_LOOP

        MOV     DL,' '                  ;Add another space before characters
        CALL    WRITE_CHAR
        MOV     CX,16
        POP     BX                      ;Get back offset into SECTOR
        ~~XOR     BX,BX~~
ASCII_LOOP:
        MOV     DL,SECTOR[BX]
        CALL    WRITE_CHAR
        INC     BX
        LOOP    ASCII_LOOP

        POP     DX
        POP     CX
        POP     BX
        RET

        ~~MOV     AH,4Ch                  ;Return to DOS~~
        ~~INT     21h~~
DISP_LINE       ENDP

        END     DISP_HALF_SECTOR
```

The changes are all fairly straightforward. In DISP_LINE, we have added a PUSH BX and POP BX around the HEX_LOOP, because we want to reuse the initial offset in ASCII_LOOP. We have also added PUSH and POP instructions to save and restore all the registers used within DISP_LINE. Actually, DISP_LINE is almost done; the only changes left are to add spaces and graphics characters so you will have an attractive display; those will come later.

When you link the files (after assembling Disp_sec and Cursor), remember that you now have three files: Disp_sec, Video_io, and Cursor. Disp_sec should be first in this list. You should see a display like the one in Figure 14-7 when you run the new Disp_sec.exe.

You will have more files before you are finished. Let's move on to the next chapter, where you will read a sector directly from the disk before dumping half a sector.



```
C>disp_sec
10 10 10 10 10 10 10 10 10 10 10 10 10 10 10 10  ................
11 11 11 11 11 11 11 11 11 11 11 11 11 11 11 11  ................
12 12 12 12 12 12 12 12 12 12 12 12 12 12 12 12  ................
13 13 13 13 13 13 13 13 13 13 13 13 13 13 13 13  ................
14 14 14 14 14 14 14 14 14 14 14 14 14 14 14 14  ................
15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15  ................
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16  ................
17 17 17 17 17 17 17 17 17 17 17 17 17 17 17 17  ................
18 18 18 18 18 18 18 18 18 18 18 18 18 18 18 18  ................
19 19 19 19 19 19 19 19 19 19 19 19 19 19 19 19  ................
1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A 1A  ................
1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B 1B  ................
1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C 1C  ................
1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D 1D  ................
1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E 1E  ................
1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F 1F  ................

C>_
```

Figure 14-7: Output from Disp_sec.

Summary

You know more about the different memory modes for addressing memory and registers in the 80x86 microprocessor. You learned about indirect memory addressing, which you first used to read 16 bytes of memory.

You also used indirect memory addressing in several programs you wrote in this chapter, starting with your program to print 16 hex numbers on the screen. These 16 numbers came from an area in memory labeled SECTOR, which was expanded a bit later so you could display a memory dump for 256 bytes—half a sector.

At last, you have begun to see dumps of the screen as they appear on your display, rather than as they are set in type. We will use these screen dumps to more advantage in the following chapters.

CHAPTER

Dumping a Disk Sector

In this chapter you will learn how to use NMake to simplify building your programs (you will just type NMake to rebuild). You will also learn how to read a disk sector, how to refer to variables defined in other files using EXTRN and PUBLIC, and how to load the address of a variable into a register. Finally, you will learn how to keep the size of your EXE file down even when you define large variables.

Files altered: DISP_SEC.ASM, DISK_IO.ASM

Disk files: DISP_S15.ASM, DISK_I15.ASM

15

Topics Covered

Now that you have a program that dumps 256 bytes of memory, you can add some procedures to read a sector from the disk and place it in memory starting at SECTOR. Then, your dump procedures will dump the first half of this disk sector.

Making Life Easier

With the three source files from the last chapter, life becomes somewhat complicated. Did you change all three of the files you were working on, or just two? You probably assembled all three, rather than checked to see if you made any changes since the last assemble. (Of course, if you are using the Programmer's Workbench, you did not have to keep track of which files you changed because it does that for you. This is just one of the advantages of using PWB.

Assembling all of your source files when you have only changed one of them is rather slow and tedious, and will become even slower as Dskpatch grows in size. What you would really like to do is assemble only the files you have changed.

Fortunately, both the assemblers covered in this book (MASM and Turbo Assembler) allow you to do just that. Microsoft and Borland provide programs called Make and NMake, respectively, that do exactly what you want. To use them, you create a file called Makefile that tells NMake, or Make, how to do its work. Then all you have to do is type:

```
C>NMAKE
```

NMake then assembles only the files you have changed.

Note

If you are using Borland's Make, you will type MAKE instead of NMAKE.

If you are using MASM 5 or earlier, you will have to type MAKE MAKEFILE because Make did not automatically look for a specific file, whereas both NMake and Borland's Make look for the file MAKEFILE.

The file you create, Makefile, tells NMake what files depend on which other files. Every time you change a file, DOS updates the modify time for this file (you can see this in the DIR display). NMake simply looks at both the ASM and OBJ versions of a file. If the ASM version has a more recent modify time than the OBJ version, NMake knows it needs to assemble that file again.

There is one caveat we need to point out. NMake will work correctly only if DOS' date and time are correct. This may not be the case if your computer's C-MOS clock has a dead battery.

Format of the NMake File

The format for Makefile that we will use with NMake is fairly simple. The format is as follows:

Listing 15-1 The NMake File MAKEFILE

```
disp_sec.exe:   disp_sec.obj video_io.obj cursor.obj
        link disp_sec video_io cursor;

disp_sec.obj:   disp_sec.asm
        ml /c disp_sec.asm

video_io.obj:   video_io.asm
        ml /c video_io.asm

cursor.obj:     cursor.asm
        ml /c cursor.asm
```

Each entry has a file name on the left (before the colon) and one or more file names on the right. If any of the files on the right (such as DISP_SEC.ASM in the first line) are more recent than the first file (DISP_SEC.OBJ), NMake will execute all the indented commands that appear on the following lines. Enter these lines into the file Makefile (without an extension) and make a small change to DISP_SEC.ASM. Then type the following:

If you are using a version of Make from MASM 5 or earlier, the first two lines must be at the end of the file, rather than the beginning.

```
C>NMAKE
```

If you are using Borland's Make just type MAKE. Type MAKE MAKEFILE if you are using MASM 5 or earlier and you will see something like the following:

```
C>nmake

Microsoft (R) Program Maintenance Utility   Version 1.13
Copyright (c) Microsoft Corp 1988-91. All rights reserved.

        ml /c disp_sec.asm
Microsoft (R) Macro Assembler Version 6.00
Copyright (c) Microsoft Corp 1981-1991. All rights reserved.

 Assembling: disp_sec.asm
        link disp_sec video_io cursor;

Microsoft (R) Segmented-Executable Linker  Version 5.13
Copyright (c) Microsoft Corp 1984-1991. All rights reserved.


C>
```

NMake has done the minimum amount of work necessary to rebuild your program. If you have an older version of the Microsoft Macro Assembler that doesn't include Make, you will find this program worth the price of an upgrade. You will get a nice replacement for Debug, too. It is called CodeView, and it will be covered later in this chapter.

Patching up Disp_sec

Disp_sec, as we left it, included a version of DISP_HALF_SECTOR, which you used as a test procedure and the main procedure. Now, you will change DISP_HALF_SECTOR to an ordinary procedure so it can be called from a procedure named READ_SECTOR. The test procedure to read a disk sector will be in Disk_io.

First, modify Disp_sec to make it a file of procedures, just as you did with Video_io. Change the END DISP_HALF_SECTOR to just END, since your main procedure will now be in Disk_io. Then you will need to remove the

.STACK and DOSSEG directives near the top of Disp_sec.asm, again because you are moving these to a different file.

Since we plan to read a sector into memory starting at SECTOR, there is no need to supply test data. We can replace all the 16 DB statements after SECTOR with two lines which reserves 8,192 bytes for storing a sector.

```
        PUBLIC  SECTOR
SECTOR  DB      8192 DUP (0)
```

Recall our earlier statement that sectors are 512 bytes long. Why, then, do you need such a large storage area? In the old days of DOS, some hard disk companies used large sectors with larger hard disks (300 megabytes, for example) instead of adding more sections, which was needed for versions of DOS before 3.31. These large sector sizes are by no means common these days, but we still want to be certain that you do not read in a sector that is too large to fit into the memory you have reserved for SECTOR. So, in the interest of safety, we have reserved 8,192 bytes for SECTOR. In the rest of this book, with the exception of SECTOR, which we will cover soon, we will assume that sectors are only 512 bytes long.

Now what you need is a new version of DISP_HALF_SECTOR. The old version is nothing more than a test procedure that we used to test DISP_LINE. In the new version, we will want to supply an offset into the sector so you can display 256 bytes, starting anywhere in the sector. Among other things, this means you could dump the first half, the last half, or the middle 256 bytes. Once again, DX supplies this offset. The new, and final, version of DISP_HALF_SECTOR in Disp_sec is as follows:

Listing 15-2 The Final Version of DISP_HALF_SECTOR in DISP_SEC.ASM (Complete Listing in DISP_S15.ASM)

```
        PUBLIC  DISP_HALF_SECTOR
        EXTRN   SEND_CRLF:PROC
;-----------------------------------------------------------------------;
;  This procedure displays half a sector (256 bytes)                    ;
;                                                                       ;
; On entry:     DS:DX   Offset into sector, in bytes -- should be      ;
;                       multiple of 16.                                 ;
;                                                                       ;
```

continues

Listing 15-2 continued

```
; Uses:          DISP_LINE, SEND_CRLF                                   ;
;-------------------------------------------------------------------;
DISP_HALF_SECTOR        PROC
~~        MOV     AX,DGROUP                 ;Put data segment into AX~~
~~        MOV     DS,AX                     ;Set DS to point to data~~

~~        XOR     DX,DX                     ;Start at beginning of SECTOR~~
        PUSH    CX
        PUSH    DX
        MOV     CX,16                     ;Display 16 lines
HALF_SECTOR:
        CALL    DISP_LINE
        CALL    SEND_CRLF
        ADD     DX,16
        LOOP    HALF_SECTOR
        POP     DX
        POP     CX
        RET

~~        MOV     AH,4Ch                    ;Return to DOS~~
~~        INT     21h~~
DISP_HALF_SECTOR        ENDP
```

Let's move on to our procedure to read a sector.

Reading a Sector

In this first version of READ_SECTOR we will deliberately ignore errors, such as having no disk in the disk drive. This is not good practice, but this isn't the final version of READ_SECTOR. We won't be able to cover error handling in this book, but you will find error-handling procedures in the version of Dskpatch on the disk that is included with this book. For now we just want to read a sector from the disk. The test version of the file DISK_IO.ASM is as follows:

Listing 15-3 The New File DISK_IO.ASM (DISK_I15.ASM)

```
DOSSEG
.MODEL  SMALL

.STACK

.DATA

        EXTRN   SECTOR:BYTE

.CODE

        EXTRN   DISP_HALF_SECTOR:PROC
;-----------------------------------------------------------------------;
; This procedure reads the first sector on disk A and dumps the first  ;
; half of this sector.                                                  ;
;-----------------------------------------------------------------------;
READ_SECTOR     PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        MOV     AL,0                    ;Disk drive A (number 0)
        MOV     CX,1                    ;Read only 1 sector
        MOV     DX,0                    ;Read sector number 0
        LEA     BX,SECTOR               ;Where to store this sector
        INT     25h                     ;Read the sector
        POPF                            ;Discard flags put on stack by DOS
        XOR     DX,DX                   ;Set offset to 0 within SECTOR
        CALL    DISP_HALF_SECTOR        ;Dump the first half

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
READ_SECTOR     ENDP

        END     READ_SECTOR
```

There are three new instructions in this procedure. The first moves the *address*, or offset, of SECTOR (from the start of the DGROUP data group created by .DATA) into the BX register; LEA stands for *Load Effective Address*.

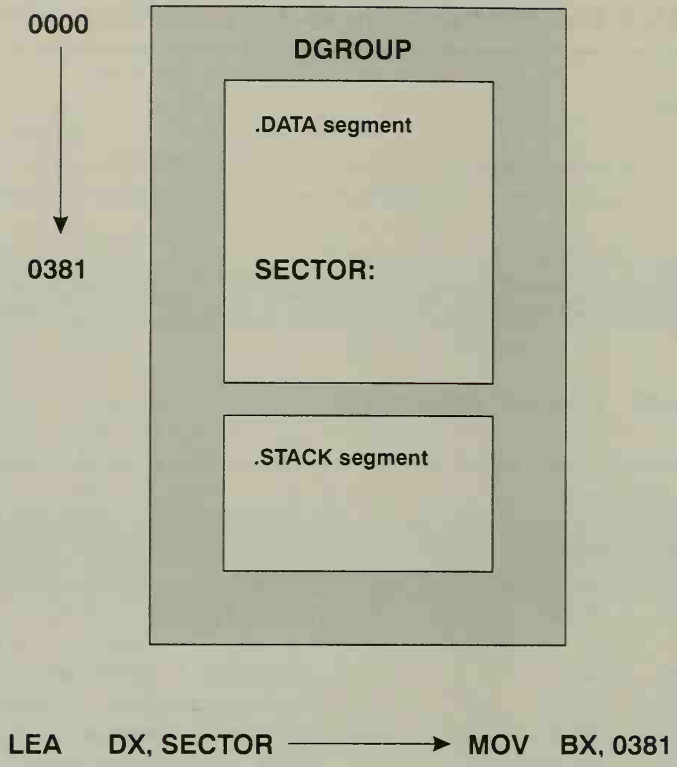


Figure 15-1: LEA loads the effective address.

```
LEA       BX,SECTOR
```

After this LEA instruction, DS:BX contains the full address of SECTOR, and DOS uses this address for the second new instruction, the INT 25h call, as you will see after a few more words about SECTOR. Actually, LEA loads the offset into the BX register without setting the DS register. You have to ensure that DS is pointing to the correct segment.

SECTOR is not in the same source file as READ_SECTOR. It is over in DISP_SEC.ASM. You tell the assembler where it is by using the EXTRN directive as follows and in Figure 15-2.

```
.DATA, EXTRN, SECTOR:BYTE
```

This set of instructions tells the assembler that SECTOR is defined in the data segment created by .DATA, and that it is defined in another source file, and that SECTOR is a variable of bytes (rather than words). We will be using such

EXTRNs often in following chapters; it is the way we use the same variables in a number of source files. Just be careful to define variables in only one place.

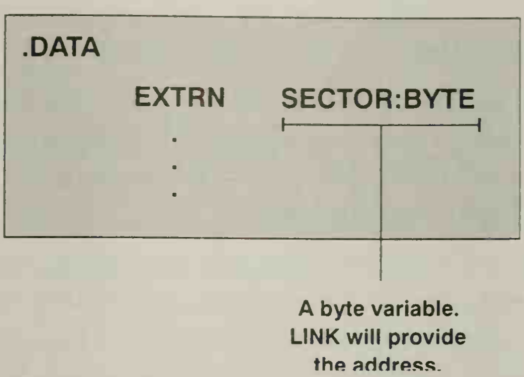


Figure 15-2: The EXTRN directive.

Let's return to the INT 25h instruction. INT 25h is a special function call to DOS for reading sectors from a disk. When DOS receives a call from INT 25h, it uses the information in the registers as follows:

AL	Drive number (0=A, 1=B, and so on)
CX	Number of sectors to read at one time
DX	Number of the first sector to read (the first sector is 0)
DS:BX	Transfer address: where to write the sectors read

The number in the AL register determines the drive from which DOS will read sectors. If AL = 0, DOS reads from drive A.

Note

All versions of DOS since COMPAQ DOS 3.31 and DOS 4.0 support hard disks larger than 32M bytes by changing the way the INT 25h function call works. This isn't a problem for reading from a floppy disk, as we are doing in this book, but it can be if you want to use Dskpatch on a hard disk. You will find the code to read from such hard disks in the advanced version of Dskpatch, included on the disk.

DOS can read more than one sector with a single call, and it reads the number of sectors given by CX. Here, we set CX to one so DOS will read just one sector of 512 bytes.

You set DX to zero, so DOS will read the very first sector on the disk. You can change this number if you want to read a different sector; later on, we will.

DS:BX is the full address for the area in memory where you want DOS to store the sector(s) it reads. In this case, you have set DS:BX to the address of SECTOR so you can call DISP_HALF_SECTOR to dump the first half of the first sector read from the disk in drive A.

Finally, you will notice a POPF instruction immediately following the INT 25h. As mentioned before, the 80x86 has a register called the status register that contains the various flags, like the zero and carry flags. POPF is a special POP instruction that pops a word into the status register. Why do you need this POPF instruction?

The INT 25h instruction first pushes the status registers, then the return address onto the stack. When DOS returns from this INT 25h, it leaves the status register on the stack. DOS does this so it can set the carry flag on return if there was a disk error, such as trying to read from drive A: with no disk in the drive. We won't be checking for errors in this book, but we have to remove the status register from the stack—hence the POPF instruction.

INT 25h, along with INT 24h which writes a disk sector, are the only DOS routines that leave the status register on the stack.

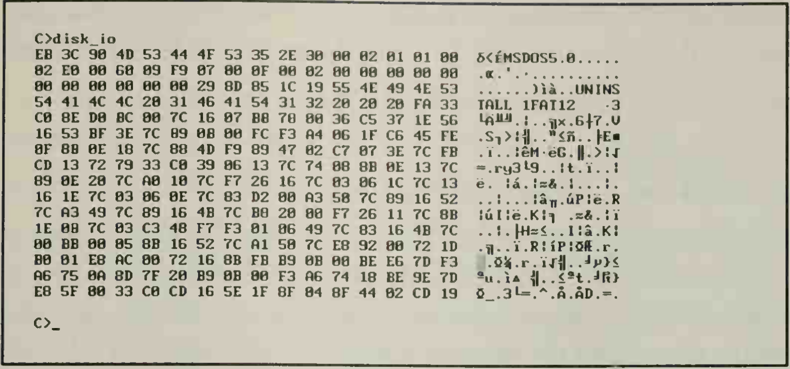
Now you can assemble DISK_IO.ASM and reassemble DISP_SEC.ASM. Then, link the four files Disk_io, Disp_sec, Video_io, and Cursor, with Disk_io listed first. Or, if you have NMake (or Borland's Make), add the following two lines to the end of your Makefile:

```
disk_io.obj:    disk_io.asm
        ml /c disk_io.asm
```

Change the first two lines to the following:

```
disk_io.exe:    disk_io.obj disp_sec.obj video_io.obj cursor.obj
        link disk_io disp_sec video_io cursor;
```

After you create your EXE version of Disk_io, you should see a display something like Figure 15-3 (remember to put a disk in drive A before you run Disk_io).



```
C>disk_io
EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00  δ<ÉMSDOS5.0.....
02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00  .α.'.·..........
00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53  ......)ìà..UNINS
54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33  TALL 1FAT12   ·3
C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56  └Ä╨╜.|..╗x.6┼7.V
16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE  .S┐>|╣..ⁿ≤ñ..╞E■
0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB  .ï..|êM·ëG.║.>|√
CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C  ═.ry3└9..|t.ï..|
89 0E 20 7C A0 18 7C F7 26 16 7C 03 06 1C 7C 13  ë. |á.|≈&.|...|.
16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52  ..|...|â╥.úP|ë.R
7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B  |úI|ë.K|╕ .≈&.|ï
1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C  ..|.├H≈≤..I|â.K|
00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D  .╗..ï.R|íP|ΦÆ.r.
B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3  ░.Φ¼.r.ï√╣..╛µ}≤
A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D  ªu.ì⌂ ╣..≤ªt.╛₧}
E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19  Φ_.3└═.^.Å.ÅD.═.

C>_
```

Figure 15-3: Screen dump from DISK_IO.COM.

The .DATA? Directive

If you look back at the definition of SECTOR in Disp_sec.asm, you will see that we reserved 8,192 bytes of zeros, which means that you have to reserve room in the DISK_IO.EXE file on your disk.

```
C>DIR DISK_IO.EXE

 Volume in drive C has no label
 Volume Serial Number is 191C-8737
 Directory of C:\SOURCE\ASM

DISK_IO  EXE      8920 10-17-92   3:28p
        1 file(s)       8920 bytes
                     1310720 bytes free

C>
```

As you can see, Disk_io.exe is 8,920 bytes long, which is mostly filled with zeros. That's a lot of space to reserve just for zeros, especially since you do not care what's in SECTOR before you read a sector into memory. So does SECTOR really need to take space on the disk? Nope.

There is another directive, .DATA?, that allows you to define memory variables that take space in memory, but not on the disk. You can do this by telling the assembler that you do not care what value a memory variable has. Change the three lines in DISP_SEC that define SECTOR to the following:

```
.DATA?

        PUBLIC  SECTOR
        SECTOR  DB      8192 DUP (?)
```

There are two changes here. First, there is a ? after the .DATA directive, which tells the assembler you are about to define variables that do not have initial values and, therefore, do not need to take space in the disk file. Second there is a ? rather than a 0 for the value of each byte in SECTOR. The DUP (?) tells the assembler that you do not care what value each byte has.

Warning

You need to define variables in the .DATA? section with DUP (?). If you define any variables with a value (such as VAR DB 0), or if you use VAR DB ?, the assembler will reserve room in the EXE file for *all* the variables in .DATA?. In other words, put all the variables that have initial values into .DATA, and all variables with DUP (?) in .DATA?.

After making these changes, rebuild Disk_io.exe. It should now be only 727 bytes long. The .DATA? directive allows you to keep your programs quite small on the disk.

We will come back later to add more to Disk_io; we have enough for now. In the next chapter, we will build a nicer sector display by adding some graphics characters to the display, and then adding a few more pieces of information.

Summary

Now that you have four different source files, Dskpatch is becoming somewhat more involved. In this chapter, we looked at the program NMake, which helps make life simpler by assembling only the files you have changed.

We also wrote a new procedure, READ_SECTOR. It is in a different source file from SECTOR, so you used an EXTRN definition in DISK_IO.ASM to tell the assembler about SECTOR, and let it know that SECTOR is a byte variable.

You also learned about the LEA (Load Effective Address) instruction, which you used to load the address of SECTOR into the BX register.

DISK_IO uses a new INT number, INT 25h, to read sectors from a disk to memory. You used INT 25h to read one sector into your memory variable, SECTOR, so you could dump it on the screen with DISP_HALF_SECTOR.

You also learned about the POPF instruction to pop a word off the stack and into the status register. You used this instruction to remove the flags which DOS did not remove from the stack when it returned from INT 25h.

The half-sector display is not very attractive yet. In the next chapter you will use some of the graphics characters available on the PC to make it more aesthetically pleasing.

C H A P T E R

Enhancing the Sector Display

In this chapter you will continue building the Dskpatch program, adding lines around the display, hex offsets along the left side, and the top. Finally, you will learn two new instructions: LODSB (Load String Byte) and CLD (Clear Direction), which are useful for working with strings of data.

Files altered: DISP_SEC.ASM, VIDEO_IO.ASM, DISK_IO.ASM

Disk files: DISP_S16.ASM, VIDEO_I16.ASM, DISK_I16.ASM

16

Topics Covered

You have come to the last chapter in Part II. Everything covered so far has been applicable to DOS and the 80x86. In Part III, you will begin to write procedures that work more closely with your computer's screen.

But before we move on, we will use this chapter to add several more procedures to Video_io. We will also modify DISP_LINE in Disp_sec. All modifications and additions will be to the display. Most of them will be to improve the appearance of the display, but one will add new information. It will add numbers on the left that act like the addresses in Debug's dump. Let's begin with graphics.

Adding Graphics Characters

The PC has a number of line-drawing characters you can use to draw boxes around various parts of the dump display. You will draw one box around the hex dump and another around the ASCII dump.

Enter the following definitions near the top of the file DISP_SEC.ASM, between the .MODEL directive and the .DATA? directive. Leave one or two blank lines before and after these definitions:

Listing 16-1 Add to the Top of DISP_SEC.ASM (No Complete Listing on Disk)

```
;----------------------------------------------------------------------;
; Graphics characters for border of sector.                            ;
;----------------------------------------------------------------------;
VERTICAL_BAR    EQU     0BAh
HORIZONTAL_BAR  EQU     0CDh
UPPER_LEFT      EQU     0C9h
UPPER_RIGHT     EQU     0BBh
LOWER_LEFT      EQU     0C8h
LOWER_RIGHT     EQU     0BCh
TOP_T_BAR       EQU     0CBh
BOTTOM_T_BAR    EQU     0CAh
TOP_TICK        EQU     0D1h
BOTTOM_TICK     EQU     0CFh
```

These are the definitions for the graphics, line-drawing characters. Notice that we put a zero before each hex number so the assembler will know these are numbers, rather than labels.

We could just as easily have written hex numbers instead of these definitions in the procedure, but the definitions make the procedure easier to understand. For example, compare the following two instructions:

```
MOV     DL,VERTICAL_BAR
MOV     DL,0BAh
```

Most people find the first instruction clearer.

Now, here is the new DISP_LINE procedure to separate the different parts of the display with the VERTICAL_BAR character, number 186 (0BAh). As before, additions are shown against a gray background.

Listing 16-2 Changes to DISP_LINE in DISP_SEC.ASM

```
DISP_LINE       PROC
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                   ;Offset is more useful in BX
                                        ;Write separator
        MOV     DL,' '
        CALL    WRITE_CHAR
        MOV     DL,VERTICAL_BAR         ;Draw left side of box
        CALL    WRITE_CHAR
        MOV     DL,' '
        CALL    WRITE_CHAR
                                        ;Now write out 16 bytes
        MOV     CX,16                   ;Dump 16 bytes
        PUSH    BX                      ;Save the offset for ASCII_LOOP
HEX_LOOP:
        MOV     DL,SECTOR[BX]           ;Get 1 byte
        CALL    WRITE_HEX               ;Dump this byte in hex
        MOV     DL,' '                  ;Write a space between numbers
        CALL    WRITE_CHAR
        INC     BX
        LOOP    HEX_LOOP
```

continues

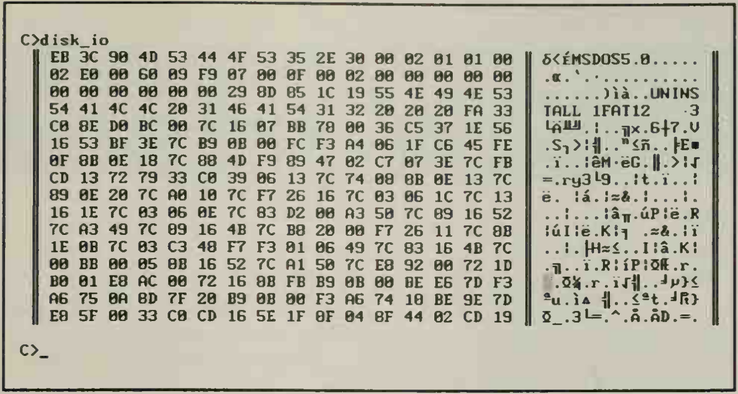
Listing 16-2 continued

```
        MOV     DL,VERTICAL_BAR         ;Write separator
        CALL    WRITE_CHAR
        MOV     DL,' '                  ;Add another space before characters
        CALL    WRITE_CHAR

        MOV     CX,16
        POP     BX                      ;Get back offset into SECTOR
ASCII_LOOP:
        MOV     DL,SECTOR[BX]
        CALL    WRITE_CHAR
        INC     BX
        LOOP    ASCII_LOOP

        MOV     DL,' '                  ;Draw right side of box
        CALL    WRITE_CHAR
        MOV     DL,VERTICAL_BAR
        CALL    WRITE_CHAR

        POP     DX
        POP     CX
        POP     BX
        RET
DISP_LINE       ENDP
```

Use NMake to build the new Disk_io, or assemble this new version of Disp_sec and link your four files (remember to place Disk_io first in the list of files following the LINK command). You'll see nice double bars separating the display into two parts, as you can see in Figure 16-1.

Adding Addresses to the Display

Now try something a bit more challenging. Add the hex addresses down the left side of the display. These numbers will be the offset from the beginning of the sector, so the first number will be 00, the next 10, then 20, and so on.



```
C>disk_io
 ║ EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00 ║ δ<ÉMSDOS5.0..... ║
 ║ 02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00 ║ .α.`.·.......... ║
 ║ 00 00 00 00 00 00 29 8D 05 1C 19 55 4E 49 4E 53 ║ ......)ìà..UNINS ║
 ║ 54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33 ║ TALL 1FAT12   ·3 ║
 ║ C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56 ║ └Ä╨╜.¦..╗x.6┼7.V ║
 ║ 16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE ║ .S┐>¦╣..ⁿ≤ñ..╞E■ ║
 ║ 0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB ║ .ï..¦êM·ëG.╟.>¦√ ║
 ║ CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C ║ ═.ry3└9..¦t.ï..¦ ║
 ║ 89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13 ║ ë. ¦á.¦≈&.¦...¦. ║
 ║ 16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52 ║ ..¦...¦â╥.úP¦ë.R ║
 ║ 7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B ║ ¦úI¦ë.K¦╕ .≈&.¦ï ║
 ║ 1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C ║ ..¦.├H≈≤..I¦â.K¦ ║
 ║ 00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D ║ .╗..ï.R¦íP¦ΦÆ.r. ║
 ║ B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3 ║ ░.Φ¼.r.ï√╣..╛µ}≤ ║
 ║ A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 10 BE 9E 7D ║ ªu.ì⌂ ╣..≤ªt.╛₧} ║
 ║ E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19 ║ Φ_.3└═.^.Å.ÅD.═. ║

C>_
```

Figure 16-1: Disk_io with vertical bars added.

The process is fairly simple, since you already have the procedure WRITE_HEX for writing a number in hex. But you do have a problem in dealing with a sector 512 bytes long: WRITE_HEX prints only two-digit hex numbers, whereas you need three hex digits for numbers greater than 255.

Here is the solution. Since your numbers will be between zero and 511 (0h to 1FFh), the first digit will either be a space, if the number (such as BCh) is below 100h, or it will be a 1. So, if the number is larger than 255, simply print a 1 followed by the hex number for the lower byte. Otherwise, you will print a space first. Additions to DISP_LINE that will print this leading three-digit hex number are as follows:

Listing 16-3 Additions to DISP_LINE in DISP_SEC.ASM

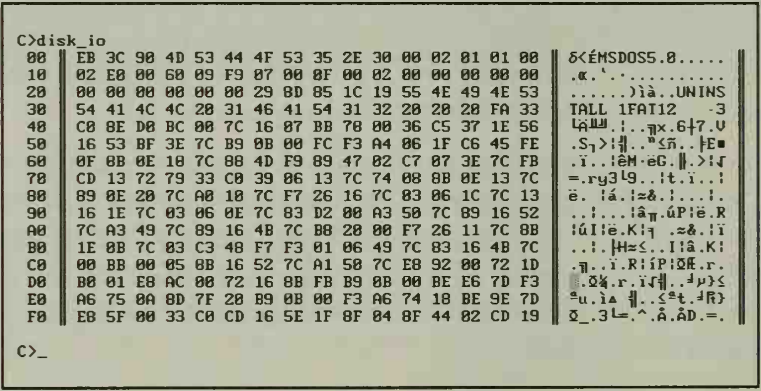
```
DISP_LINE       PROC
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                   ;Offset is more useful in BX
        MOV     DL,' '
                                        ;Write offset in hex
        CMP     BX,100h                 ;Is the first digit a 1?
        JB      WRITE_ONE               ;No, white space already in DL
        MOV     DL,'1'                  ;Yes, then place '1' into DL for output
```

continues

Listing 16-3 continued

```
WRITE_ONE:
        CALL    WRITE_CHAR
        MOV     DL,BL                   ;Copy lower byte into DL for hex output
        CALL    WRITE_HEX
                                        ;Write separator
        MOV     DL,' '
        CALL    WRITE_CHAR
        MOV     DL,VERTICAL_BAR         ;Draw left side of box
                .
                .
                .
```

The results are shown in Figure 16-2.



```
C>disk_io
 00 ║ EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00 ║ δ<ÉMSDOS5.0..... ║
 10 ║ 02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00 ║ .α.`.·.......... ║
 20 ║ 00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53 ║ ......)ìà..UNINS ║
 30 ║ 54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33 ║ TALL 1FAT12   ·3 ║
 40 ║ C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56 ║ └Ä╨╝.¦..╗x.6┼7.V ║
 50 ║ 16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE ║ .S┐>¦╣..ⁿ≤ñ..╞E■ ║
 60 ║ 0F 8B 0E 10 7C 88 4D F9 89 47 02 C7 07 3E 7C FB ║ .ï..¦êM·ëG.╟.>¦√ ║
 70 ║ CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C ║ ═.ry3└9..¦t.ï..¦ ║
 80 ║ 89 0E 20 7C A0 10 7C F7 26 16 7C 83 06 1C 7C 13 ║ ë. ¦á.¦≈&.¦â..¦. ║
 90 ║ 16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52 ║ ..¦...¦â╥.úP¦ë.R ║
 A0 ║ 7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B ║ ¦úI¦ë.K¦╕ .≈&.¦ï ║
 B0 ║ 1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C ║ ..¦.├H≈≤..I¦â.K¦ ║
 C0 ║ 00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D ║ .╗..ï.R¦íP¦ΦÆ.r. ║
 D0 ║ B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3 ║ ░.Φ¼.r.ï√╣..╛µ}≤ ║
 E0 ║ A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D ║ ªu.ì⌂ ╣..≤ªt.╛₧} ║
 F0 ║ E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19 ║ Φ_.3└═.^.Å.ÅD.═. ║

C>_
```

Figure 16-2: Disk_io after adding hex addresses on the left.

You are getting closer to the full display. But on the screen, your display is not quite centered. You need to move it to the right by about three spaces. By making this one last change, you have finished your version of DISP_LINE.

You could make the change by calling WRITE_CHAR three times with a space character, but you won't. Instead, you will add another procedure called WRITE_CHAR_N_TIMES, to Video_io. As its name implies, this procedure writes one character N times. That is, you place the number N into the CX register and the character code into DL, and call WRITE_CHAR_N_TIMES to write N copies of the character whose ASCII code you placed in DL. Then you will be able to write three spaces by placing 3 into CX and 20h (the ASCII code for a space) into DL. The procedure to add to VIDEO_IO.ASM is as follows:

Listing 16-4 Procedure Added to VIDEO_IO.ASM

```
        PUBLIC  WRITE_CHAR_N_TIMES
;-----------------------------------------------------------------------;
; This procedure writes more than one copy of a character               ;
;                                                                       ;
; On entry:     DL      Character code                                  ;
;               CX      Number of times to write the character          ;
;                                                                       ;
; Uses:         WRITE_CHAR                                              ;
;-----------------------------------------------------------------------;
WRITE_CHAR_N_TIMES      PROC
        PUSH    CX
N_TIMES:
        CALL    WRITE_CHAR
        LOOP    N_TIMES
        POP     CX
        RET
WRITE_CHAR_N_TIMES      ENDP
```

You can see how simple this procedure is, because you already have WRITE_CHAR. If you are wondering why we bothered to write a procedure for something so simple, it is because the program Dskpatch is much clearer when we call WRITE_CHAR_N_TIMES rather than write a short loop to print multiple copies of a character. In addition, this procedure will be used several times again.

The changes to DISP_LINE to add three spaces on the left of your display are shown in Listing 16-5. Make the changes to DISP_SEC.ASM:

Listing 16-5 Changes to DISP_LINE in DISP_SEC.ASM

```
        PUBLIC  DISP_LINE
        EXTRN   WRITE_HEX:PROC
        EXTRN   WRITE_CHAR:PROC
        EXTRN   WRITE_CHAR_N_TIMES:PROC
;---------------------------------------------------------------------;
;  This procedure displays one line of data, or 16 bytes, first in hex, ;
;  then in ASCII.                                                      ;
;                                                                      ;
; On entry:     DS:DX   Offset into sector, in bytes                   ;
;                                                                      ;
; Uses:         WRITE_CHAR, WRITE_HEX, WRITE_CHAR_N_TIMES              ;
; Reads:        SECTOR                                                 ;
;---------------------------------------------------------------------;
DISP_LINE       PROC
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                   ;Offset is more useful in BX
        MOV     DL,' '
        MOV     CX,3                    ;Write 3 spaces before line
        CALL    WRITE_CHAR_N_TIMES
                                        ;Write offset in hex
        CMP     BX,100h                 ;Is the first digit a 1?
        JB      WRITE_ONE               ;No, white space already in DL
        MOV     DL,'1'                  ;Yes, place '1' into DL for output
WRITE_ONE:
                   .
                   .
                   .
```

You made changes in three places. First, you had to add an EXTRN statement for WRITE_CHAR_N_TIMES because the procedure is in Video_io, and not in this file. You also changed the comment block, to show that you use this new procedure. The third change, the two lines that use WRITE_CHAR_N_TIMES, is quite straightforward and needs no explanation.

Try this new version of the program to see how the display is now centered. Next we will move on to add more features to the display—the top and bottom lines of the boxes.

Adding Horizontal Lines

Adding horizontal lines to the display is not quite as simple as it sounds, because we have a few special cases to think about. We have the ends, where the lines must go around corners, and we also have T-shaped junctions at the top and bottom of the division between the hex and ASCII windows.

We could write a long list of instructions (with WRITE_CHAR_N_TIMES) to create the horizontal lines, but we won't. We have a shorter way. We will introduce another procedure, called WRITE_PATTERN, which will write a pattern on the screen. Then, all we will need is a small area of memory to hold a description of each pattern. Using this new procedure, we can also add tick marks easily to subdivide the hex window, as you will see when we finish this section.

WRITE_PATTERN uses two entirely new instructions, LODSB and CLD. We will describe them after you see more about WRITE_PATTERN and how you describe a pattern. Right now, enter the following procedure into the file VIDEO_IO.ASM:

Listing 16-6 Add This Procedure to VIDEO_IO.ASM (Complete Listing in VIDEO_16.ASM)

```
        PUBLIC  WRITE_PATTERN
;----------------------------------------------------------------------;
; This procedure writes a line to the screen, based on data in the     ;
; form                                                                 ;
;                                                                      ;
;       DB      {character, number of times to write character}, 0     ;
; Where {x} means that x can be repeated any number of times           ;
;                                                                      ;
; On entry:     DS:DX   Address of the pattern to draw                 ;
;                                                                      ;
; Uses:         WRITE_CHAR_N_TIMES                                     ;
;----------------------------------------------------------------------;
WRITE_PATTERN   PROC
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
```

continues

Listing 16-6 continued

```
        PUSHF                           ;Save the direction flag
        CLD                             ;Set direction flag for increment
        MOV     SI,DX                   ;Move offset into SI register for LODSB
PATTERN_LOOP:
        LODSB                           ;Get character data into AL
        OR      AL,AL                   ;Is it the end of data (0h)?
        JZ      END_PATTERN             ;Yes, return
        MOV     DL,AL                   ;No, set up to write character N times
        LODSB                           ;Get the repeat count into AL
        MOV     CL,AL                   ;And put in CX for WRITE_CHAR_N_TIMES
        XOR     CH,CH                   ;Zero upper byte of CX
        CALL    WRITE_CHAR_N_TIMES
        JMP     PATTERN_LOOP
END_PATTERN:
        POPF                            ;Restore direction flag
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
WRITE_PATTERN   ENDP
```

Before you see how this procedure works, we will show you how to write data for patterns. You will place the data for the top-line pattern into the file Disp_sec, which is where you will use it. To this end, you will add another procedure called INIT_SEC_DISP to initialize the sector display by writing the half-sector display. Then you will modify READ_SECTOR to call your INIT_SEC_DISP procedure. First, place the following data before the .DATA? where you defined SECTOR (in DISP_SEC.ASM):

Listing 16-7 Additions to DISP_SEC.ASM

```
.DATA

TOP_LINE_PATTERN        LABEL   BYTE
        DB      ' ',7
        DB      UPPER_LEFT,1
        DB      HORIZONTAL_BAR,12
        DB      TOP_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      TOP_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      TOP_TICK,1
        DB      HORIZONTAL_BAR,12
        DB      TOP_T_BAR,1
        DB      HORIZONTAL_BAR,18
        DB      UPPER_RIGHT,1
        DB      0
BOTTOM_LINE_PATTERN     LABEL   BYTE
        DB      ' ',7
        DB      LOWER_LEFT,1
        DB      HORIZONTAL_BAR,12
        DB      BOTTOM_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      BOTTOM_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      BOTTOM_TICK,1
        DB      HORIZONTAL_BAR,12
        DB      BOTTOM_T_BAR,1
        DB      HORIZONTAL_BAR,18
        DB      LOWER_RIGHT,1
        DB      0

.DATA?

SECTOR  DB      8192 DUP (?)
```

Note that you put all the new data into .DATA rather than .DATA? because you need to set values for all these variables.

Each DB statement contains part of the data for one line. The first byte is the character to print; the second byte tells WRITE_PATTERN how many times to repeat that character. For example, the top line starts with seven blank spaces, followed by one upper-left-corner character, followed by twelve horizontal-bar characters, and so on. The last DB is a solitary hex zero, which marks the end of the pattern.

We will continue the modifications and show you the result before we discuss the inner workings of WRITE_PATTERN. Here is the test version of INIT_SEC_DISP. This procedure writes the top-line pattern, the half-sector display, and finally the bottom-line pattern. Place it in the file DISP_SEC.ASM, just before DISP_HALF_SECTOR as follows:

Listing 16-8 Add This Procedure to DISP_SEC.ASM

```
        PUBLIC  INIT_SEC_DISP
        EXTRN   WRITE_PATTERN:PROC, SEND_CRLF:PROC
;----------------------------------------------------------------------;
; This procedure initializes the half-sector display.                  ;
;                                                                      ;
; Uses:         WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR             ;
; Reads:        TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN                  ;
;----------------------------------------------------------------------;
INIT_SEC_DISP   PROC
        PUSH    DX
        LEA     DX,TOP_LINE_PATTERN
        CALL    WRITE_PATTERN
        CALL    SEND_CRLF
        XOR     DX,DX                   ;Start at the beginning of the sector
        CALL    DISP_HALF_SECTOR
        LEA     DX,BOTTOM_LINE_PATTERN
        CALL    WRITE_PATTERN
        POP     DX
        RET
INIT_SEC_DISP   ENDP
```

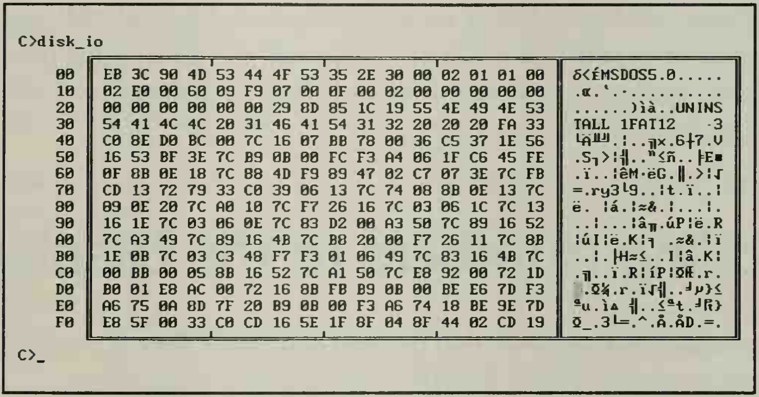
By using the LEA instruction to load an address into the DX register, WRITE_PATTERN knows where to find the pattern data.

Finally, we need to make a small change to READ_SECTOR in the file DISK_IO.ASM. This will enable you to call INIT_SECTOR_DISP, rather than WRITE_HALF_SECTOR_DISP, so that a full box will be drawn around the half-sector display.

Listing 16-9 Changes to READ_SECTOR in DISK_IO.ASM (Complete Listing in DISK_I16.ASM)

```
        EXTRN   INIT_SEC_DISP:PROC
;----------------------------------------------------------------------;
; This procedure reads the first sector on disk A and dumps the first  ;
; half of this sector.                                                 ;
;----------------------------------------------------------------------;
READ_SECTOR     PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data
        MOV     AL,0                    ;Disk drive A (number 0)
        MOV     CX,1                    ;Read only 1 sector
        MOV     DX,0                    ;Read sector number 0
        LEA     BX,SECTOR               ;Where to store this sector
        INT     25h                     ;Read the sector
        POPF                            ;Discard flags put on stack by DOS
~~        XOR     DX,DX                   ;Set offset to 0 within SECTOR~~
        CALL    INIT_SEC_DISP           ;Dump the first half

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
READ_SECTOR     ENDP
```

That's all you need to draw the top and bottom lines for the sector display. Assemble and link all these files using NMake (remember to assemble the three files you changed if you are not using NMake) and give it a try. Figure 16-3 shows the output you now have.



```
C>disk_io

  00 ║ EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00 ║ δ<ÉMSDOS5.0.....
  10 ║ 02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00 ║ .α.`.·..........
  20 ║ 00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53 ║ ......)ìà..UNINS
  30 ║ 54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33 ║ TALL 1FAT12   ·3
  40 ║ C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56 ║ └Ä╨╝.¦..╗x.6┼7.V
  50 ║ 16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE ║ .S┐>¦╣..ⁿ≤ñ..╞E■
  60 ║ 0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB ║ .ï..¦êM·ëG.╟.>¦√
  70 ║ CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C ║ =.ry3└9..¦t.ï..¦
  80 ║ 89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13 ║ ë. ¦á.¦≈&.¦...¦.
  90 ║ 16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52 ║ ..¦...¦â╥.úP¦ë.R
  A0 ║ 7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B ║ ¦úI¦ë.K¦╕ .≈&.¦ï
  B0 ║ 1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C ║ ..¦.├H≈≤..I¦â.K¦
  C0 ║ 00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D ║ .╗..ï.R¦íP¦ΦÆ.r.
  D0 ║ B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3 ║ ░.Φ¼.r.ï√╣..╛µ}≤
  E0 ║ A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D ║ ªu.ì⌂ ╣..≤ªt.╛₧}
  F0 ║ E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19 ║ Φ_.3└═.^.Å.ÅD.═.

C>_
```

Figure 16-3: The Disk_io display with closed boxes.

Let's see how WRITE_PATTERN works. As mentioned, it uses two new instructions. LODSB stands for *Load String Byte*, which is one of the string instructions—specially designed instructions that work with strings of characters. That is not quite what you are doing here, but the 80x86 doesn't care whether you are dealing with a string of characters or just numbers, so LODSB suits the purposes just fine.

LODSB moves (loads) a single byte into the AL register from the memory location given by DS:SI, a register pair you have not used before. You already set DS in READ_SECTOR to point to the data. Before the LODSB instruction, you moved the offset into the SI register with the instruction MOV SI,DX.

The LODSB instruction is somewhat like the MOV instruction, but more powerful. With one LODSB instruction, the 80x86 moves one byte into the AL register and then either increments or decrements the SI register. Incrementing the SI register points to the following byte in memory; decrementing the register points to the previous byte in memory.

Incrementing is exactly what you want to do. You want to go through the pattern, one byte at a time, starting at the beginning. That is what the LODSB instruction does because you used the other new instruction, CLD (*Clear Direction Flag*) to clear the direction flag. If you had set the direction flag, the LODSB instruction would decrement the SI register instead. We will use the LODSB instruction in a few other places in Dskpatch, always with the direction flag cleared, to increment.

Aside from LODSB and CLD, notice that we also used the PUSHF and POPF instructions to save and restore the flag register. We did this just in case we later decide to use the direction flag in a procedure that calls WRITE_PATTERN.

Adding Numbers to the Display

We are almost through with Part II of this book now. We will create one more procedure, then it is on to Part III, and bigger and better things.

Right now, notice that the display lacks a row of numbers across the top. Such numbers—00 01 02 03 and so forth—would allow you to sight down the columns to find the address for any byte. So, let's write a procedure to print this row of numbers. Add the following procedure, WRITE_TOP_HEX_NUMBERS to DISP_SEC.ASM just after INIT_SEC_DISP.

Listing 16-10 Add This Procedure to DISP_SEC.ASM

```
        EXTRN   WRITE_CHAR_N_TIMES:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC
        EXTRN   WRITE_HEX_DIGIT:PROC, SEND_CRLF:PROC
;-----------------------------------------------------------------------;
; This procedure writes the index numbers (0 through F) at the top of   ;
; the half-sector display.                                              ;
;                                                                       ;
; Uses:         WRITE_CHAR_N_TIMES, WRITE_HEX, WRITE_CHAR               ;
;               WRITE_HEX_DIGIT, SEND_CRLF                              ;
;-----------------------------------------------------------------------;
WRITE_TOP_HEX_NUMBERS   PROC
        PUSH    CX
        PUSH    DX
        MOV     DL,' '                  ;Write 9 spaces for left side
        MOV     CX,9
        CALL    WRITE_CHAR_N_TIMES
        XOR     DH,DH                   ;Start with 0
HEX_NUMBER_LOOP:
        MOV     DL,DH
        CALL    WRITE_HEX
        MOV     DL,' '
        CALL    WRITE_CHAR
        INC     DH
        CMP     DH,10h                  ;Done yet?
        JB      HEX_NUMBER_LOOP

        MOV     DL,' '                  ;Write hex numbers over ASCII window
        MOV     CX,2
        CALL    WRITE_CHAR_N_TIMES
        XOR     DL,DL
HEX_DIGIT_LOOP:
        CALL    WRITE_HEX_DIGIT
        INC     DL
        CMP     DL,10h
        JB      HEX_DIGIT_LOOP
        CALL    SEND_CRLF
        POP     DX
        POP     CX
        RET
WRITE_TOP_HEX_NUMBERS   ENDP
```

Modify INIT_SEC_DISP (also in DISP_SEC.ASM) as follows so it calls WRITE_TOP_HEX_NUMBERS before it writes the rest of the half-sector display.

Listing 16-11 Changes to INIT_SEC_DISP in DISP_SEC.ASM (Complete Listing in DISP_S16.ASM)

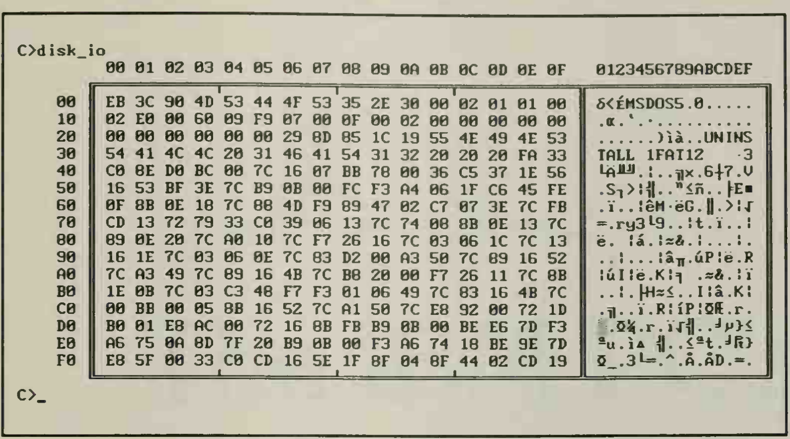
```
; Uses:          WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR              ;
;                WRITE_TOP_HEX_NUMBERS                                   ;
; Reads:         TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN                   ;
;-----------------------------------------------------------------------;
INIT_SEC_DISP   PROC
        PUSH    DX
        CALL    WRITE_TOP_HEX_NUMBERS
        LEA     DX,TOP_LINE_PATTERN
        CALL    WRITE_PATTERN
        CALL    SEND_CRLF
        XOR     DX,DX                   ;Start at the beginning of the sector
        CALL    DISP_HALF_SECTOR
        LEA     DX,BOTTOM_LINE_PATTERN
        CALL    WRITE_PATTERN
        POP     DX
        RET
INIT_SEC_DISP   ENDP
```

Now you have a complete half-sector display, as shown in Figure 16-4.

There are still some differences between this display and the final version. We will change WRITE_CHAR so it will print all 256 characters the PC can display, and then we will clear the screen and center this display vertically, using the ROM BIOS routines inside the PC.

Summary

You have done a lot of work on the Dskpatch program, by adding new procedures, changing old ones, and moving them from one source file to another. From now on, if you find yourself losing track of what you are doing, refer to the complete listing of Dskpatch in Appendix B. The listing there is the final version, but you will probably see enough resemblances to help you along.



```
C>disk_io
           00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F    0123456789ABCDEF

    00     EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00    δ<ÉMSDOS5.0.....
    10     02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00    .α.`.·..........
    20     00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53    ......)ìà..UNINS
    30     54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33    TALL 1FAT12   ·3
    40     C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56    └Ä╨╝.¦..╗x.6┼7.V
    50     16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE    .S┐>¦╣..ⁿ≤ñ..╞E■
    60     0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB    .ï..¦êM·ëG.╟.>¦√
    70     CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C    ═.ry3└9..¦t.ï..¦
    80     89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13    ë. ¦á.¦≈&.¦...¦.
    90     16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52    ..¦...¦â╥.úP¦ë.R
    A0     7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B    ¦úI¦ë.K¦╕ .≈&.¦ï
    B0     1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C    ..¦.├H≈≤..I¦â.K¦
    C0     00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D    .╗..ï.R¦íP¦ΦÆ.r.
    D0     B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3    ░.Φ¼.r.ï√╣..╛µ}≤
    E0     A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D    ªu.ì⌂ ╣..≤ªt.╛₧}
    F0     E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19    Φ_.3└═.^.Å.ÅD.═.

C>_
```

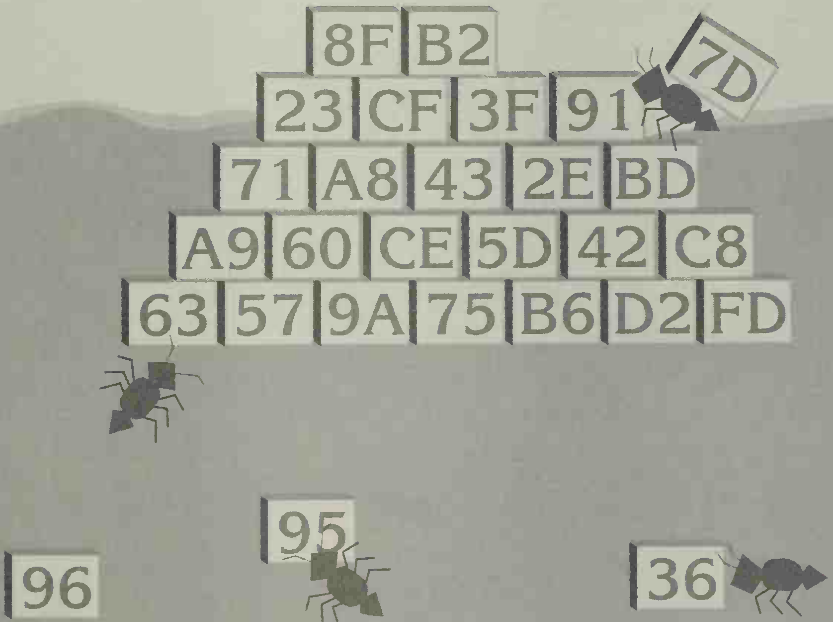
Figure 16-4: A complete half-sector display.

Most of the changes in this chapter did not rely on tricks, just hard work. But you did learn two new instructions: LODSB and CLD. LODSB is one of the string instructions that allow you to use one instruction to do the work of several. You used LODSB in WRITE_PATTERN to read consecutive bytes from the pattern table, always loading a new byte into the AL register. CLD clears the direction flag, which sets the direction for increment. Each following LODSB instruction loads the next byte from memory.

In the next part of this book you will learn about the PC's ROM BIOS routines. They will save you a lot of time.

P A R T III

The IBM PC's ROM BIOS



C H A P T E R

The ROM BIOS Routines

In this chapter you will learn how to use the INT 10h ROM BIOS routines to clear your screen and move the cursor to any location on the screen. You will rewrite Dskpatch so that it clears the screen before it draws. Then you will rewrite the code so that READ_SECTOR uses memory variables, rather than hard-wired numbers. Finally, you will add a status line to the top of Dskpatch's screen.

Files altered: DSKPATCH.ASM, DISP_SEC.ASM, CURSOR.ASM, VIDEO_IO.ASM, DISK_IO.ASM

Disk files: DSKPAT17.ASM, DISP_S17.ASM, CURSOR17.ASM, VIDEO_17.ASM, DISK_I17.ASM

17

Topics Covered

The ROM BIOS Display Routines

Clearing the Screen

Moving the Cursor

Rewiring Variable Usage

Writing the Header

Summary

Computer chips, or ICs (*Integrated Circuits*), known as ROMs (*Read-Only Memory*) are located inside your computer. One of these ROMs contains a number of routines, very much like procedures that provide all the basic routines for doing input and output to several different parts of your PC. Because this ROM provides routines for performing input and output at a very low level, it is frequently referred to as the BIOS, for Basic Input Output System. DOS uses the ROM BIOS for such activities as sending characters to the screen and reading and writing to the disk, and you are free to use the ROM BIOS routines in your programs.

We will concentrate on the BIOS routines needed for Dskpatch. Among them is a set for video display, which includes a number of functions you couldn't reach without working directly with the hardware.

The ROM BIOS Display Routines

We refer to the elements of the ROM BIOS as routines in order to distinguish them from procedures. Procedures are used with a CALL instruction; whereas you call routines with INT instructions, not CALLs. You will use an INT 10h instruction, for example, to call the ROM BIOS' video I/O routines just as you used an INT 21h instruction to call routines in DOS.

INT 10h calls the video I/O routines in the ROM BIOS; other numbers call other routines, but you won't see any of them. The INT 10h routines provide all the functions you need outside of DOS. (DOS calls one of the other ROM BIOS routines when you ask for a sector from the disk.)

In this chapter, we will use ROM BIOS routines to add two new procedures to Dskpatch: one to clear the screen, and the other to move the cursor to any screen location you chose. Both are very useful functions, but neither is available directly through DOS. Hence, you will use the ROM BIOS routines to do the job. Later, you will see even more interesting things that you can do with these ROM routines. Begin by using INT 10h to clear the screen before you display the half sector.

The INT 10h instruction is your entry to a number of different functions. Recall that, when you used the DOS INT 21h instruction, you selected a particular function by placing its function number in the AH register. You select an INT 10h function in the same way—by placing the appropriate function number in the AH register. A full list of these functions is given in Table 17-1.

Table 17-1 INT 10h Functions

(AH)=0	**Set the display mode.** The AL register contains the mode number.	
		Text Modes
	(AL)=0	40 by 25, black and white mode
	(AL)=1	40 by 25, color
	(AL)=2	80 by 25, black and white
	(AL)=3	80 by 25, color
	(AL)=7	80 by 25, monochrome display adapter
		Graphic Mode
	(AL)=4	320 by 200, color
	(AL)=5	320 by 200, black and white
	(AL)=6	640 by 200, black and white
(AH)=1	**Set the cursor size.**	
	(CH)	Starting scan line of the cursor. The top line is 0 on both the monochrome and color graphics displays, and the bottom line is 7 for the color graphics adapter and 13 for the monochrome adapter. Valid range: 0 to 31.
	(CL)	Last scan line of the cursor.
	The power-on setting for the color graphics adapter is CH=6 and CL=7. The setting for the monochrome display is CH=11 and CL=12.	

continues

Table 17-1 continued

(AH)=2	**Set the cursor position.**		
	(DH,DL)	Row, column of new cursor position; the upper left corner is (0,0).	
	(BH)	Page number. This is the number of the display page. The color-graphics adapter has room for several display pages, but most programs use page 0.	
(AH)=3	**Read the cursor position.**		
	(BH)	Page number	
	On exit	(DH,DL)	Row, column of cursor
		(CH,CL)	Cursor size
(AH)=4	**Read light pen position.**		
(AH)=5	**Select active display page.**		
	(AL)	New page number (from 0 to 7 for modes 0 and 1; from 0 to 3 for modes 2 and 3).	
(AH)=6	**Scroll up.**		
	(AL)	Number of lines to blank at the bottom of the window. Normal scrolling blanks one line. Set to zero to blank entire window.	
	(CH,CL)	Row, column of upper left corner of window.	
	(DH,DL)	Row, column of lower right corner of window.	
	(BH)	Display attribute to use for blank lines.	
	(AH)=7	**Scroll down.**	
	Same as scroll up (function 6), but lines are left blank at the top of the window instead of the bottom.		

(AH)=8	**Read attribute and character under the cursor.**	
	(BH)	Display page (text modes only).
	(AL)	Character read.
	(AH)	Attribute of character read (text modes only).
(AH)=9	**Write attribute and character under the cursor.**	
	(BH)	Display page (text modes only).
	(CX)	Number of times to write character and attribute on screen.
	(AL)	Character to write.
	(BL)	Attribute to write.
(AH)=10	**Write character under cursor** (with normal attribute).	
	(BH)	Display page.
	(CX)	Number of times to write character.
	(AL)	Character to write.
(AH)=11 to 13	**Various graphics functions.**	
(AH)=14	**Write teletype.** Write one character to the screen and move the cursor to the next position.	
	(AL)	Character to write.
	(BL)	Color of character (graphics mode only).
	(BH)	Display page (text mode).
(AH)=15	**Return current video state.**	
	(AL)	Display mode currently set.
	(AH)	Number of characters per line.
	(BH)	Active display pages.

Clearing the Screen

We will use the INT 10h function number 6, *Scroll Active Page Up*, to clear the screen. You don't actually want to scroll the screen, but this function also doubles as a clear-screen function. Enter the following procedure into the file CURSOR.ASM.

Listing 17-1 Procedure Added to CURSOR.ASM

```
        PUBLIC  CLEAR_SCREEN
;----------------------------------------------------------------------;
; This procedure clears the entire screen.                             ;
;----------------------------------------------------------------------;
CLEAR_SCREEN    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        XOR     AL,AL                   ;Blank entire window
        XOR     CX,CX                   ;Upper left corner is at (0,0)
        MOV     DH,24                   ;Bottom line of screen is line 24
        MOV     DL,79                   ;Right side is at column 79
        MOV     BH,7                    ;Use normal attribute for blanks
        MOV     AH,6                    ;Call for SCROLL-UP function
        INT     10h                     ;Clear the window
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CLEAR_SCREEN    ENDP
```

It appears that INT 10h function number 6 needs quite a lot of information, even though all you want to do is clear the display. This function is rather powerful. It can actually clear any rectangular part of the screen (called a window). You have to set the window to the entire screen by setting the first and last lines to 0 and 24 and setting the columns to 0 and 79. The routine can also clear the screen to all white (for use with black characters) or all black (for use with white characters). You want the latter, which is specified with the instruction MOV BH,7. Setting AL to 20, the number of lines to scroll, tells this routine to clear the window, rather than to scroll it.

Now you need to modify the test procedure, READ_SECTOR, to call CLEAR_SCREEN just before it starts to write the sector display. We did not place this CALL in INIT_SEC_DISP, because you will want to use INIT_SEC_DISP to rewrite just the half-sector display, without affecting the rest of the screen.

To modify READ_SECTOR, add an EXTRN declaration for CLEAR_SCREEN and insert the CALL to CLEAR_SCREEN. Make the following changes in the file DISK_IO.ASM:

Listing 17-2 Changes to READ_SECTOR in DISK_IO.ASM (Complete Listing in DISK_I17.ASM)

```
        EXTRN   INIT_SEC_DISP:PROC, CLEAR_SCREEN:PROC
;-------------------------------------------------------------------;
; This procedure reads the first sector on disk A and dumps the first   ;
; half of this sector.                                              ;
;-------------------------------------------------------------------;
READ_SECTOR     PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        MOV     AL,0                    ;Disk drive A (number 0)
        MOV     CX,1                    ;Read only 1 sector
        MOV     DX,0                    ;Read sector number 0
        LEA     BX,SECTOR               ;Where to store this sector
        INT     25h                     ;Read the sector
        POPF                            ;Discard flags put on stack by DOS
        CALL    CLEAR_SCREEN
        CALL    INIT_SEC_DISP           ;Dump the first half

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
READ_SECTOR     ENDP
```

Note where the cursor is located and then run Disk_io. The screen will clear, and Disk_io will start writing the half-sector display wherever the cursor happened to be before you ran the program—probably at the bottom of the screen.

Even though the screen was cleared, we didn't mention anything about moving the cursor back to the top. In BASIC, the CLS command does two things—it clears the screen and then moves the cursor to the top of the screen. Our procedure doesn't do that; you will have to move the cursor yourself.

Moving the Cursor

The INT 10h function number 2 sets the cursor position, which you will use in the GOTO_XY subroutine. You can use GOTO_XY to move the cursor anywhere on the screen (such as to the top after a clear). Enter this procedure into the file CURSOR.ASM:

Listing 17-3 Procedure Added to CURSOR.ASM (Complete Listing in CURSOR17.ASM)

```
        PUBLIC  GOTO_XY
;-------------------------------------------------------------------;
; This procedure moves the cursor                                   ;
;                                                                   ;
; On entry:     DH      Row (Y)                                     ;
;               DL      Column (X)                                  ;
;-------------------------------------------------------------------;
GOTO_XY         PROC
        PUSH    AX
        PUSH    BX
        MOV     BH,0                    ;Display page 0
        MOV     AH,2                    ;Call for SET CURSOR POSITION
        INT     10h
        POP     BX
        POP     AX
        RET
GOTO_XY         ENDP
```

You will use GOTO_XY in a revised version of INIT_SEC_DISP to move the cursor to the second line just before you write the half-sector display. Modifications made to INIT_SEC_DISP in DISP_SEC.ASM are as follows:

Listing 17-4 Changes to INIT_SEC_DISP in DISP_SEC.ASM

```
        PUBLIC  INIT_SEC_DISP
        EXTRN   WRITE_PATTERN:PROC, SEND_CRLF:PROC
        EXTRN   GOTO_XY:PROC
;----------------------------------------------------------------------;
;  This procedure initializes the half-sector display.                 ;
;                                                                      ;
; Uses:         WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR             ;
;               WRITE_TOP_HEX_NUMBERS, GOTO_XY                         ;
; Reads:        TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN                  ;
;----------------------------------------------------------------------;
INIT_SEC_DISP   PROC
        PUSH    DX
        XOR     DL,DL                   ;Move cursor to beginning
        MOV     DH,2                    ;of 3rd line
        CALL    GOTO_XY
        CALL    WRITE_TOP_HEX_NUMBERS
        LEA     DX,TOP_LINE_PATTERN
                    .
                    .
                    .
```

If you try the new procedure now, you will see that the half-sector display is nicely centered.

It is easier to work with the screen when you have the ROM BIOS routines. In the next chapter, you will use another routine in the ROM BIOS to improve WRITE_CHAR so that it will write any character to the screen. But before continuing, let's make some other changes to your program. After the changes are made, you will finish up by adding a procedure called WRITE_HEADER. This procedure will write a status line at the top of the screen to show the current disk drive and sector number.

Rewiring Variable Usage

There is much that needs to revamped before creating WRITE_HEADER. Many of our procedures as they are now have numbers hard-wired into them; for example, READ_SECTOR reads sector 0 on drive A. We want to place the disk-drive and sector numbers into memory variables, so more than one procedure can read them. You will need to change these procedures so that they will use memory variables.

You will begin by putting all memory variables into one file, DSKPATCH.ASM, to make your work simpler. Dskpatch.asm will be the first file in your program so the memory variables will be easy to find there. The DSKPATCH.ASM, complete with a long list of memory variables is as folows:

Listing 17-5 The New File DSKPATCH.ASM

```
DOSSEG
.MODEL  SMALL

.STACK

.DATA

        PUBLIC  SECTOR_OFFSET
;-----------------------------------------------;
; SECTOR_OFFSET is the offset of the half-      ;
; sector display into the full sector. It must  ;
; be a multiple of 16, and not greater than 256 ;
;-----------------------------------------------;
SECTOR_OFFSET   DW      0

        PUBLIC  CURRENT_SECTOR_NO, DISK_DRIVE_NO
CURRENT_SECTOR_NO       DW      0               ;Initially sector 0
DISK_DRIVE_NO           DB      0               ;Initially Drive A:

        PUBLIC  LINES_BEFORE_SECTOR, HEADER_LINE_NO
        PUBLIC  HEADER_PART_1, HEADER_PART_2
;-----------------------------------------------;
; LINES_BEFORE_SECTOR is the number of lines    ;
; at the top of the screen before the half-     ;
; sector display.                               ;
;-----------------------------------------------;
```

```
LINES_BEFORE_SECTOR     DB      2
HEADER_LINE_NO          DB      0
HEADER_PART_1           DB      'Disk ',0
HEADER_PART_2           DB      '          Sector ',0

.DATA?

        PUBLIC  SECTOR
;-------------------------------------------;
; The entire sector (up to 8192 bytes) is   ;
; stored in this part of memory.            ;
;-------------------------------------------;
SECTOR  DB      8192 DUP (?)

.CODE

        EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
        EXTRN   INIT_SEC_DISP:PROC
DISK_PATCH      PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        CALL    CLEAR_SCREEN
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
DISK_PATCH      ENDP

        END     DISK_PATCH
```

The main procedure, DISK_PATCH, calls three other procedures. You have seen them all before and soon you will rewrite both READ_SECTOR and INIT_SEC_DISP to use the variables you just placed into the data segment.

Before using Dskpatch, you need to modify Disp_sec to replace the definition of SECTOR with an EXTRN. You also need to alter Disk_io, to change READ_SECTOR into an ordinary procedure you can call from Dskpatch.

Let's take SECTOR first. Because you have placed it in DSKPATCH.ASM as a memory variable, you need to change the definition of SECTOR in Disp_sec to an EXTRN declaration. Make the following changes in DISP_SEC.ASM:

Listing 17-6 Changes to DISP_SEC.ASM

```
.DATA?

        EXTRN   SECTOR:BYTE
~~        PUBLIC  SECTOR~~
~~SECTOR  DB      8192 DUP(?)~~
```

Next you will rewrite the file DISK_IO.ASM so that it contains only procedures and so that READ_SECTOR uses memory variables (not hard-wired numbers) for the sector and disk-drive numbers. The new version of DISK_IO.ASM is as follows:

Listing 17-7 Changes to DISK_IO.ASM

```
~~DOSSEG~~
~~.MODEL  SMALL~~

~~.STACK~~

.DATA

        EXTRN   SECTOR:BYTE
        EXTRN   DISK_DRIVE_NO:BYTE
        EXTRN   CURRENT_SECTOR_NO:WORD

.CODE

        PUBLIC  READ_SECTOR
~~        EXTRN   INIT_SEC_DISP:PROC, CLEAR_SCREEN:PROC~~
;-------------------------------------------------------------------;
; This procedure reads one sector (512 bytes) into SECTOR.          ;
;                                                                   ;
; Reads:        CURRENT_SECTOR_NO, DISK_DRIVE_NO                    ;
; Writes:       SECTOR                                              ;
;-------------------------------------------------------------------;
```

```
READ_SECTOR      PROC
        ~~MOV     AX,DGROUP                  ;Put data segment into AX~~
        ~~MOV     DS,AX                      ;Set DS to point to data~~
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,DISK_DRIVE_NO           ;Drive number
        MOV     CX,1                       ;Read only 1 sector
        MOV     DX,CURRENT_SECTOR_NO       ;Logical sector number
        LEA     BX,SECTOR                  ;Where to store this sector
        INT     25h                        ;Read the sector
        POPF                               ;Discard flags put on stack by DOS
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
        ~~CALL    CLEAR_SCREEN~~
        ~~CALL    INIT_SEC_DISP              ;Dump the first half~~

        ~~MOV     AH,4Ch                     ;Return to DOS~~
        ~~INT     21h~~
READ_SECTOR      ENDP
        END
```

This new version of Disk_io uses the memory variables DISK_DRIVE_NO and CURRENT_SECTOR_NO as the disk-drive and sector numbers for the sector to read. Because these variables are already defined in DSKPATCH.ASM, you won't have to change Disk_io when you start reading different sectors from other disk drives. If you are using the NMake program to rebuild DSKPATCH.COM, you will need to make some additions to your Make file named Makefile. The additions are as follows:

Listing 17-8 The New Version of MAKEFILE

```
dskpatch.exe:   dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj
        link dskpatch disk_io disp_sec video_io cursor;
```

continues

Listing 17-8 continued

```
dskpatch.obj:   dskpatch.asm
        ml /c dskpatch.asm

disk_io.obj:    disk_io.asm
        ml /c disk_io.asm

disp_sec.obj:   disp_sec.asm
        ml /c disp_sec.asm

video_io.obj:   video_io.asm
        ml /c video_io.asm

cursor.obj:     cursor.asm
        ml /c cursor.asm
```

Remember that if you are using Make from MASM 5 or earlier, the first two lines shown here must be at the end of your Makefile. If you are not using NMake, be sure to reassemble all three files you have changed (Dskpatch, Disk_io, and Disp_sec) and link your five files with Dskpatch listed first:

```
LINK DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR;
```

You have made quite a few changes, so test Dskpatch and make sure that it works correctly before moving on.

Writing the Header

Now that we have converted the hard-wired numbers into direct references to memory variables, we can write the procedure WRITE_HEADER to write a status line, or header, at the top of the screen. The header will look like this:

```
Disk A          Sector 0
```

WRITE_HEADER will use WRITE_DECIMAL to write the current sector number in decimal. It will also write two strings of characters, *Disk* and *Sector* (each followed by a blank space), and a disk letter, such as A. To begin, place the following procedure in DISP_SEC.ASM:

Listing 17-9 Procedure Added to DISP_SEC.ASM (Complete Listing in DISP_S17.ASM)

```
        PUBLIC  WRITE_HEADER
.DATA
        EXTRN   HEADER_LINE_NO:BYTE
        EXTRN   HEADER_PART_1:BYTE
        EXTRN   HEADER_PART_2:BYTE
        EXTRN   DISK_DRIVE_NO:BYTE
        EXTRN   CURRENT_SECTOR_NO:WORD
.CODE
        EXTRN   WRITE_STRING:PROC, WRITE_DECIMAL:PROC
        EXTRN   GOTO_XY:PROC
;-------------------------------------------------------------------;
; This procedure writes the header with disk-drive and sector number.   ;
;                                                                   ;
; Uses:         GOTO_XY, WRITE_STRING, WRITE_CHAR, WRITE_DECIMAL    ;
; Reads:        HEADER_LINE_NO, HEADER_PART_1, HEADER_PART_2        ;
;               DISK_DRIVE_NO, CURRENT_SECTOR_NO                    ;
;-------------------------------------------------------------------;
WRITE_HEADER    PROC
        PUSH    DX
        XOR     DL,DL                   ;Move cursor to header line number
        MOV     DH,HEADER_LINE_NO
        CALL    GOTO_XY
        LEA     DX,HEADER_PART_1
        CALL    WRITE_STRING
        MOV     DL,DISK_DRIVE_NO
        ADD     DL,'A'                  ;Print drives A, B, ...
        CALL    WRITE_CHAR
        LEA     DX,HEADER_PART_2
        CALL    WRITE_STRING
        MOV     DX,CURRENT_SECTOR_NO
        CALL    WRITE_DECIMAL
        POP     DX
        RET
WRITE_HEADER    ENDP
```

The procedure WRITE_STRING doesn't exist yet. As you can see, we plan to use it to write a string of characters to the screen. The two strings, HEADER_PART_1 and HEADER_PART_2, are already defined in DSKPATCH.ASM. WRITE_STRING will use DS:DX as the address for the string.

We have chosen to supply our own string-output procedure so that our strings can contain any character, including the $ which could not be printed with DOS function 9. In places where DOS uses a $ to mark the end of a string, we will use a hex 0. Enter the following procedure into VIDEO_IO.ASM:

Listing 17-10 Procedure Added to VIDEO_IO.ASM (Complete Listing in VIDEO_17.ASM)

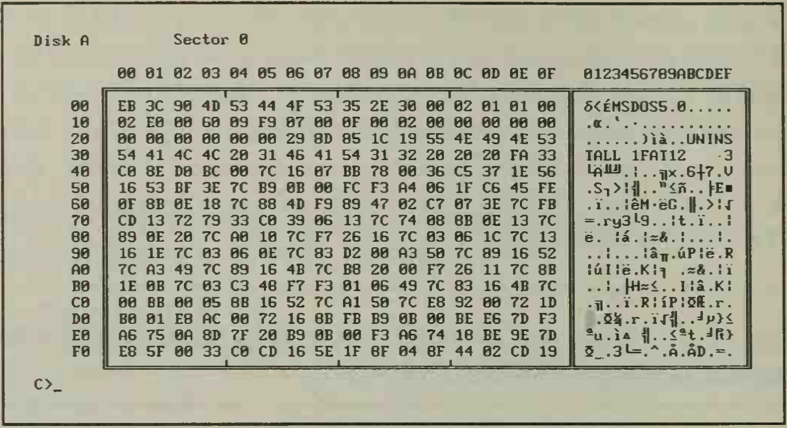
```
        PUBLIC  WRITE_STRING
;-------------------------------------------------------------------;
; This procedure writes a string of characters to the screen. The  ;
; string must end with          DB      0                           ;
;                                                                   ;
; On entry:     DS:DX   Address of the string                       ;
;                                                                   ;
; Uses:         WRITE_CHAR                                          ;
;-------------------------------------------------------------------;
WRITE_STRING    PROC
        PUSH    AX
        PUSH    DX
        PUSH    SI
        PUSHF                           ;Save direction flag
        CLD                             ;Set direction for increment
        MOV     SI,DX                   ;Place address into SI for LODSB
STRING_LOOP:
        LODSB                           ;Get a character into AL register
        OR      AL,AL                   ;Have we found the 0 yet?
        JZ      END_OF_STRING           ;Yes, we are done with the string
        MOV     DL,AL                   ;No, write character
        CALL    WRITE_CHAR
        JMP     STRING_LOOP
END_OF_STRING:
        POPF                            ;Restore direction flag
        POP     SI
        POP     DX
        POP     AX
        RET
WRITE_STRING    ENDP
```

As it stands now, WRITE_STRING will write characters with ASCII codes below 32 (the space character) as a period (.), because we don't have a version of WRITE_CHAR that can write *any* character. We will take care of that detail in the next chapter. The advantage of modular design is that you will not have to change WRITE_STRING in the process. Finally, change DISK_PATCH in DSKPATCH.ASM to include the CALL to WRITE_HEADER as follows:

Listing 17-11 Changes to DISK_PATCH in DSKPATCH.ASM (Complete Listing in DSKPAT17.ASM)

```
        EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
DISK_PATCH      PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        CALL    CLEAR_SCREEN
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
DISK_PATCH      ENDP
```

Dskpatch should now produce a display like the one in Figure 17-1.



```
Disk A          Sector 0

      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F    0123456789ABCDEF

 00   EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00    δ<ÉMSDOS5.0.....
 10   02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00    .α.`.˙..........
 20   00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53    ......)ìà..UNINS
 30   54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33    TALL 1FAT12   ·3
 40   C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56    └Ä╨╝.¦..╗x.6┼7.V
 50   16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE    .S┐>¦╣..ⁿ≤ñ..╞E■
 60   0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB    .ï..¦êM·ëG.╟.>¦√
 70   CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C    ═.ry3└9..¦t.ï..¦
 80   89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13    ë. ¦á.¦≈&.¦...¦.
 90   16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52    ..¦...¦â╥.úP¦ë.R
 A0   7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B    ¦úI¦ë.K¦╕ .≈&.¦ï
 B0   1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C    ..¦.├H≈≤..I¦â.K¦
 C0   00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D    .╗..ï.R¦íP¦ΦÆ.r.
 D0   B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3    ░.Φ¼.r.ï√╣..╛µ}≤
 E0   A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D    ªu.ì⌂ ╣..≤ªt.╛₧}
 F0   E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19    Φ_.3└═.^.Å.ÅD.═.

C>_
```

Figure 17-1: Dskpatch with the header at the top.

Summary

At last you have met the ROM BIOS routines inside of your PCs. You have already used two of these routines to help you toward your goal of a full Dskpatch program.

First you learned about INT 10h, function number 6, which you used to clear the screen. You briefly saw that this function has more uses than you will take advantage of in this book. For example, you may eventually find it helpful for scrolling portions of the screen in Dskpatch or in your own programs.

Then you used function 2 of INT 10h to move the cursor to the third line on the screen (line number 2). That is where you started writing the sector dump.

To make programs easier to work with, several procedures were rewritten so that they would use memory variables, rather than hard-wired numbers. Now you will be able to read other sectors and change the way your program works in other ways by changing a few central numbers in DSKPATCH.ASM.

Finally, you wrote the procedures WRITE_HEADER and WRITE_STRING so that a header could be placed at the top of the screen. As mentioned, an improved version of WRITE_CHAR will be written in the next chapter, replacing the dots in the ASCII window of the display with graphics characters. Thanks to modular design, you will do this without changing any of the procedures that use WRITE_CHAR.

CHAPTER

The Ultimate WRITE_CHAR

In this chapter, you will finally change WRITE_CHAR so that it can display any of the 256 characters on your PC. This won't be that difficult, using several new INT 10h functions. You will use function 9 to display a character on the screen, and you will use functions 3 and 2 to read and set the position of the cursor that you will need in order to move the cursor right one position after displaying a character.

Files altered: DISP_SEC.ASM, CURSOR.ASM, VIDEO_IO.ASM

Disk files: DISP_S18.ASM, CURSOR18.ASM, VIDEO_18.ASM

18

Topics Covered

A New WRITE_CHAR

Clearing to the End of a Line

Summary

You made good use of the ROM BIOS routines in the last chapter to clear the screen and move the cursor. But there are many more uses for the ROM BIOS, and you will see some of them in this chapter.

Using DOS alone has not enabled you to display all 256 of the characters that the PC is capable of displaying. So, in this chapter, you will write a new version of WRITE_CHAR that displays any character, thanks to another INT 10h function.

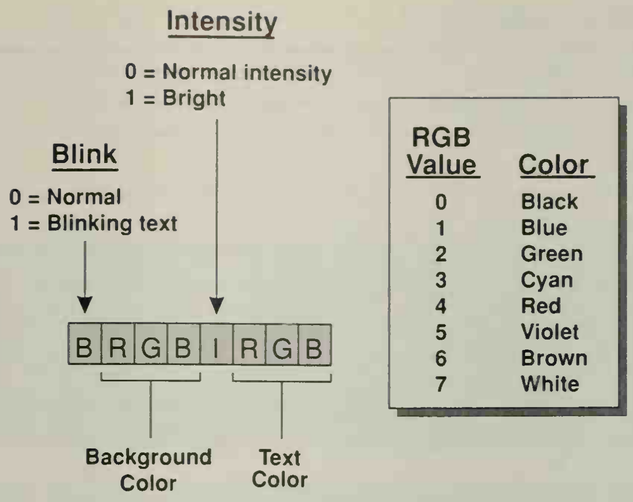
Then add another useful procedure, called CLEAR_TO_END_OF_LINE, which clears the line from the cursor to the right edge of the screen. This procedure will be put to use in WRITE_HEADER so that it will clear the rest of the line. Why is this useful? Suppose that we go from sector number 10 (two digits) to sector number 9. A zero would be left over from the 10 after you call WRITE_HEADER with the sector set to 9. CLEAR_TO_END_OF_LINE will clear this zero, as well as anything else on the remainder of the line.

A New WRITE_CHAR

The ROM BIOS function 9 for INT 10h writes a character and its *attribute* at the current cursor position. The attribute controls such features as underlining, blinking, and color as in Figure 18-1. You will use only two attributes for Dskpatch: attribute 7, which is the normal attribute; and attribute 70h, which is a foreground color of 0 and background of 7 and produces inverse video (black characters on a white background). You can set the attributes individually for each character by creating a block cursor in inverse video—we will call it a *phantom* cursor. For now, we will use the normal attribute when we write a character.

The INT 10h, function 9 writes the character and attribute at the current cursor position. Unlike DOS, it does not advance the cursor to the next character position unless it writes more than one copy of the character. This fact will be used later, in a different procedure. For now, you only want one copy of each character, so you will move the cursor yourself.

Here is the new version of WRITE_CHAR, which writes a character and then moves the cursor right one character. Make the following changes to WRITE_CHAR in the file VIDEO_IO.ASM:



RGB Value	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Violet
6	Brown
7	White

Figure 18-1: How the ROM BIOS specifies character colors.

Listing 18-1 Changes to WRITE_CHAR in VIDEO_IO.ASM (Complete Listing in video_18.asm)

```
        PUBLIC  WRITE_CHAR
        EXTRN   CURSOR_RIGHT:PROC
;----------------------------------------------------------------------;
; This procedure outputs a character to the screen using the ROM BIOS  ;
; routines, so that characters such as the backspace are treated as    ;
; any other character and are displayed.                               ;
;                                                                      ;
; This procedure must do a bit of work to update the cursor position.  ;
;                                                                      ;
; On entry:     DL      Byte to print on screen                        ;
;                                                                      ;
; Uses:         CURSOR_RIGHT                                           ;
;----------------------------------------------------------------------;
WRITE_CHAR      PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
```

continues

Listing 18-1 continued

```
~~        CMP     DL,32                   ;Is character before a space?~~
~~        JAE     IS_PRINTABLE            ;No, then print as is~~
~~        MOV     DL,'.'                  ;Yes, replace with a period~~
~~IS_PRINTABLE:~~
~~        MOV     AH,2                    ;Call for character output~~
~~        INT     21h                     ;Output character in DL register~~
        MOV     AH,9                    ;Call for output of character/attribute
        MOV     BH,0                    ;Set to display page 0
        MOV     CX,1                    ;Write only one character
        MOV     AL,DL                   ;Character to write
        MOV     BL,7                    ;Normal attribute
        INT     10h                     ;Write character and attribute
        CALL    CURSOR_RIGHT            ;Now move to next cursor position
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
WRITE_CHAR      ENDP
```

In reading through this procedure, you may have wondered why we included the instruction MOV BH,0. If you have a graphics display adapter (as opposed to an ancient Monochrome Display Adapter), your adapter has four text pages in normal text mode. We will only use the first page, page 0; hence the instruction to set BH to 0.

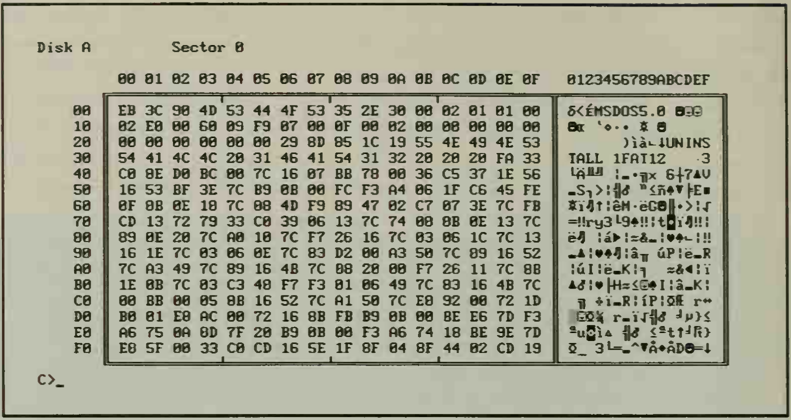
As for the cursor, WRITE_CHAR uses the procedure CURSOR_RIGHT to move the cursor right one character position or to the beginning of the next line if the movement would take the cursor past column 79. Place the following procedure into CURSOR.ASM.

Listing 18-2 Procedure Added to CURSOR.ASM

```
        PUBLIC  CURSOR_RIGHT
;----------------------------------------------------------------------;
; This procedure moves the cursor one position to the right or to the  ;
; next line if the cursor was at the end of a line.                    ;
;                                                                      ;
; Uses:           SEND_CRLF                                           ;
;---------------------------------------------------------------------;
CURSOR_RIGHT     PROC
        PUSH     AX
        PUSH     BX
        PUSH     CX
        PUSH     DX
        MOV      AH,3                     ;Read the current cursor position
        MOV      BH,0                     ;On page 0
        INT      10h                      ;Read cursor position
        MOV      AH,2                     ;Set new cursor position
        INC      DL                       ;Set column to next position
        CMP      DL,79                    ;Make sure column <= 79
        JBE      OK
        CALL     SEND_CRLF                ;Go to next line
        JMP      DONE
OK:     INT      10h
DONE:   POP      DX
        POP      CX
        POP      BX
        POP      AX
        RET
CURSOR_RIGHT     ENDP
```

CURSOR_RIGHT uses two new INT 10h functions. Function 3 reads the position of the cursor, and function 2 changes the cursor position. The procedure first uses function 3 to find the cursor position, which is returned in two bytes, the column number in DL, and the line number in DH. Then CURSOR_RIGT increments the column number (in DL) and moves the cursor. If DL was at the last column (79), the procedure sends a carriage-return/line-feed pair to move the cursor to the next line. You do not need this column 79 check in Dskpatch, but including it makes CURSOR_RIGHT a general-purpose procedure you can use in any of your own programs.

With these changes, Dskpatch should now display all 256 characters as shown in Figure 18-2. You can verify that it does by searching for a byte with a value less than 20h and seeing whether some strange character has replaced the period that value formerly produced in the ASCII window.



```
Disk A          Sector 0

        00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF
   00   EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00   δ<ÉMSDOS5.0 ☻☺☺
   10   02 E0 00 60 09 F9 07 00 0F 00 02 00 08 00 00 00   ☻α `○∙• ☼ ☻
   20   00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53         )ìà∟↓UNINS
   30   54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33   TALL 1FAT12   ·3
   40   C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56   └Ä╨╝ |▬•╗x 6┼7▲V
   50   16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE   ▬S┐>|╣♂ ⁿ≤ñ♠▼╞E■
   60   0F 8B 0E 18 7C 08 4D F9 89 47 02 C7 07 3E 7C FB   ☼ï♫↑|êM·ëG☻╟•>|√
   70   CD 13 72 79 33 C0 39 06 13 7C 74 00 8B 0E 13 7C   =‼ry3└9♠‼|t◙ï♫‼|
   80   89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13   ë♫ |á►|≈&▬|♥♠∟|‼
   90   16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52   ▬▲|♥♠♫|â╥ úP|ë▬R
   A0   7C A3 49 7C 89 16 4B 7C 08 20 00 F7 26 11 7C 8B   |úI|ë▬K|┐  ≈&◄|ï
   B0   1E 0B 7C 03 C3 40 F7 F3 01 06 49 7C 83 16 4B 7C   ▲♂|♥├H≈≤☺♠I|â▬K|
   C0   00 8B 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D    ╖ ♣ï▬R|íP|ΦÆ r↔
   D0   B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3   ░☺Φ¼ r▬ï√╣♂ ╛µ}≤
   E0   A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D   ªu◙ì⌂ ╣♂ ≤ªt↑╛₧}
   F0   E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19   Φ_ 3└═▬^▼Å♦ÅD☻═↓

C>_
```

Figure 18-2: Dskpatch with the new WRITE_CHAR.

Now onto something more interesting—writing a procedure to clear a line from the cursor position to the end.

Clearing to the End of a Line

In the last chapter, you used INT 10h, function 6, to clear the screen in the CLEAR_SCREEN procedure. At that time, we mentioned that function 6 could be used to clear any rectangular window. That capability applies even if a window is only one line high and less than one line long, so you can use function 6 to clear part of a line—to the end of the line.

The left side of the window, in this case, is the column number of the cursor, which you get with a function 3 call (also used by CURSOR_RIGHT). The right side of the window is always at column 79. You can see the details in CLEAR_TO_END_OF_LINE; place the procedure in CURSOR.ASM as follows:

Listing 18-3 Procedure Added to CURSOR.ASM (Complete Listing in CURSOR 18.ASM)

```
        PUBLIC  CLEAR_TO_END_OF_LINE
;-----------------------------------------------------------------------;
; This procedure clears the line from the current cursor position to   ;
; the end of that line.                                                 ;
;-----------------------------------------------------------------------;
CLEAR_TO_END_OF_LINE    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Read current cursor position
        XOR     BH,BH                   ; on page 0
        INT     10h                     ;Now have (X,Y) in DL, DH
        MOV     AH,6                    ;Set up to clear to end of line
        XOR     AL,AL                   ;Clear window
        MOV     CH,DH                   ;All on same line
        MOV     CL,DL                   ;Start at the cursor position
        MOV     DL,79                   ;And stop at the end of the line
        MOV     BH,7                    ;Use normal attribute
        INT     10h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CLEAR_TO_END_OF_LINE    ENDP
```

You will use this procedure in WRITE_HEADER to clear the rest of the line when you start reading other sectors (that will be covered soon). You cannot see CLEAR_TO_END_OF_LINE work with WRITE_HEADER until you add the procedures that enable you to read a different sector and update the display. But revise WRITE_HEADER now, just to get it out of the way. Make the following changes to WRITE_HEADER in DISP_SEC.ASM to call CLEAR_TO_END_OF_LINE at the end of the procedure:

Listing 18-4 Changes to WRITE_HEADER in DISP_SEC.ASM (Complete Listing in DISP_S18.ASM)

```
        PUBLIC  WRITE_HEADER
.DATA
        EXTRN   HEADER_LINE_NO:BYTE
        EXTRN   HEADER_PART_1:BYTE
        EXTRN   HEADER_PART_2:BYTE
        EXTRN   DISK_DRIVE_NO:BYTE
        EXTRN   CURRENT_SECTOR_NO:WORD
.CODE
        EXTRN   WRITE_STRING:PROC, WRITE_DECIMAL:PROC
        EXTRN   GOTO_XY:PROC, CLEAR_TO_END_OF_LINE:PROC
;----------------------------------------------------------------------;
;  This procedure writes the header with disk-drive and sector number. ;
;                                                                      ;
; Uses:         GOTO_XY, WRITE_STRING, WRITE_CHAR, WRITE_DECIMAL       ;
;               CLEAR_TO_END_OF_LINE                                   ;
; Reads:        HEADER_LINE_NO, HEADER_PART_1, HEADER_PART_2           ;
;               DISK_DRIVE_NO, CURRENT_SECTOR_NO                       ;
;----------------------------------------------------------------------;
WRITE_HEADER    PROC
        PUSH    DX
        XOR     DL,DL                   ;Move cursor to header line number
        MOV     DH,HEADER_LINE_NO
        CALL    GOTO_XY
        LEA     DX,HEADER_PART_1
        CALL    WRITE_STRING
        MOV     DL,DISK_DRIVE_NO
        ADD     DL,'A'                  ;Print drives A, B, ...
        CALL    WRITE_CHAR
        LEA     DX,HEADER_PART_2
        CALL    WRITE_STRING
        MOV     DX,CURRENT_SECTOR_NO
        CALL    WRITE_DECIMAL
        CALL    CLEAR_TO_END_OF_LINE    ;Clear rest of sector number
        POP     DX
        RET
WRITE_HEADER    ENDP
```

This revision marks both the final version of WRITE_HEADER and the completion of the file CURSOR.ASM. You are still missing several important parts of Dskpatch, though. In the next chapter, you will continue by adding the central dispatcher for keyboard commands. You will then be able to press F3 and F4 to read other sectors on the disk.

Summary

This chapter has been relatively easy, without much in the way of new information or tricks. You did learn how to use INT 10h, function number 9, in the ROM BIOS to write any character to the screen.

In the process, you also saw how to read the cursor position with INT 10h function 3 so that the cursor could be moved right one position after you wrote a character. The reason: INT 10h function 9 does not move the cursor after it writes just one character, unless it writes more than one copy of the character. Finally, you put INT 10h function 6 to work by clearing part of just one line. In the next chapter, we will get down to business again as we build the central dispatcher.

C H A P T E R

The Dispatcher

In this chapter you will build a new version of Dskpatch that can display sectors other than sector 1, using the keyboard for navigation. This chapter contains a lot of code and sets the pace for the rest of the chapters in Part III.

Files altered: DSKPATCH.ASM, DISPATCH.ASM, DISP_IO.ASM, KBD_IO.ASM, DISK_IO.ASM

Disk files: DSKPAT19.ASM, DISP_S19.ASM, DISP_I19.ASM, KBD_IO19.ASM, DISK_I19.ASM

19

Topics Covered

Building a Dispatcher

Reading Other Sectors

Philosophy of the Following Chapters

In any language it is nice to have a well-written program that does something. To really bring a program to life, we need to make it interactive. It is human nature to say, "If I do this, you do that," so we will use this chapter to add some interactivity to Dskpatch.

We will write a simple keyboard-input procedure and a central dispatcher. The dispatcher's job will be to call the correct procedure for each key pushed. For example, when you press the F3 key to read and display the previous sector, the dispatcher will call a procedure called PREVIOUS_SECTOR. To do this, you will be making many changes to Dskpatch. You will start by creating DISPATCHER, the central dispatcher, and some other procedures for display formatting. Next, you will add two new procedures, PREVIOUS_SECTOR and NEXT_SECTOR, which will be called through DISPATCHER.

Building a Dispatcher

The Dispatcher will be the central control for Dskpatch, so all keyboard input and editing will be done through it. DISPATCHER's job will be to read characters and call other procedures to do the work. You will soon see how the dispatcher does its work, but first let's see how it fits into Dskpatch.

DISPATCHER will have its own prompt line, just under the half-sector display where the cursor waits for keyboard input. You won't be able to enter hex numbers in the first version of the keyboard-input procedure, but later on you will. Here are the first modifications to DSKPATCH.ASM; these add the data for a prompt line as follows:

Listing 19-1 Additions to DATA_SEG in DSKPATCH.ASM

```
HEADER_LINE_NO          DB      0
HEADER_PART_1           DB      'Disk ',0
HEADER_PART_2           DB      '         Sector ',0
        PUBLIC  PROMPT_LINE_NO, EDITOR_PROMPT
PROMPT_LINE_NO          DB      21
EDITOR_PROMPT           DB      'Press function key, or enter'
                        DB      ' character or hex byte: ',0
```

You will add more prompts later to take care of such matters as inputting a new sector number. Your job will be made simpler by using a common procedure, WRITE_PROMPT_LINE, to write each prompt line. Each procedure that uses WRITE_PROMPT_LINE will supply it with the address of the prompt (here, the address of EDITOR_PROMPT), and then write the prompt on line 21 (because PROMPT_LINE_NO is 21). For example, this new version of DISK_PATCH (in DSKPATCH.ASM) uses WRITE_PROMPT_LINE just before it calls DISPATCHER. The new version is as follows:

Listing 19-2 Additions to DISK_PATCH in DSKPATCH.ASM (Complete Listing in DSKPAT19.ASM)

```
        EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
DISK_PATCH      PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        CALL    CLEAR_SCREEN
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        CALL    DISPATCHER

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
DISK_PATCH      ENDP
```

The dispatcher itself is a fairly simple program, but we do use some new tricks in it. The following listing is the first version of the file DISPATCH.ASM.

Listing 19-3 The New File DISPATCH.ASM (Complete Listing in DISPAT19.ASM)

```
.MODEL  SMALL

.CODE
        EXTRN   NEXT_SECTOR:PROC                        ;In DISK_IO.ASM
        EXTRN   PREVIOUS_SECTOR:PROC                    ;In DISK_IO.ASM
.DATA
;-----------------------------------------------------------------------;
; This table contains the legal extended ASCII keys and the addresses  ;
; of the procedures that should be called when each key is pressed.    ;
;                                                                       ;
; The format of the table is                                            ;
;               DB      72              ;Extended code for cursor up   ;
;               DW      OFFSET _TEXT:PHANTOM_UP                         ;
;-----------------------------------------------------------------------;
DISPATCH_TABLE  LABEL   BYTE
        DB      61                              ;F3
        DW      OFFSET _TEXT:PREVIOUS_SECTOR
        DB      62                              ;F4
        DW      OFFSET _TEXT:NEXT_SECTOR
        DB      0                               ;End of the table

.CODE

        PUBLIC  DISPATCHER
        EXTRN   READ_BYTE:PROC
;-----------------------------------------------------------------------;
; This is the central dispatcher. During normal editing and viewing,   ;
; this procedure reads characters from the keyboard and, if the char   ;
; is a command key (such as a cursor key), DISPATCHER calls the        ;
; procedures that do the actual work. This dispatching is done for     ;
; special keys listed in the table DISPATCH_TABLE, where the procedure ;
; addresses are stored just after the key names.                      ;
;                                                                       ;
; If the character is not a special key, then it should be placed      ;
; directly into the sector buffer—this is the editing mode.            ;
;                                                                       ;
; Uses:         READ_BYTE                                               ;
;-----------------------------------------------------------------------;
DISPATCHER      PROC
        PUSH    AX
        PUSH    BX
DISPATCH_LOOP:
        CALL    READ_BYTE                 ;Read character into AX
        OR      AH,AH                     ;AX = -1 if no character read, 1
                                          ; for an extended code.
        JS      DISPATCH_LOOP             ;No character read, try again
        JNZ     SPECIAL_KEY               ;Read extended code
; do nothing with the character for now
        JMP     DISPATCH_LOOP             ;Read another character

SPECIAL_KEY:
        CMP     AL,68                     ;F10--exit?
        JE      END_DISPATCH              ;Yes, leave
                                          ;Use BX to look through table
        LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
        CMP     BYTE PTR [BX],0           ;End of table?
        JE      NOT_IN_TABLE              ;Yes, key was not in the table
        CMP     AL,[BX]                   ;Is it this table entry?
        JE      DISPATCH                  ;Yes, then dispatch
        ADD     BX,3                      ;No, try next entry
        JMP     SPECIAL_LOOP              ;Check next table entry

DISPATCH:
        INC     BX                        ;Point to address of procedure
        CALL    WORD PTR [BX]             ;Call procedure
        JMP     DISPATCH_LOOP             ;Wait for another key

NOT_IN_TABLE:                             ;Do nothing, just read next char
        JMP     DISPATCH_LOOP

END_DISPATCH:
        POP     BX
        POP     AX
        RET
DISPATCHER      ENDP


        END
```

The DISPATCH_TABLE holds the extended ASCII codes for the F3 and F4 keys. Each code is followed by the address of the procedure DISPATCHER should call when it reads that particular extended code. For example, when

READ_BYTE, which is called by DISPATCHER, reads an F3 key (extended code 61), DISPATCHER calls the procedure PREVIOUS_SECTOR.

The addresses of the procedures we want DISPATCHER to call are in the dispatch table, so we used a new directive, OFFSET, to obtain them. The following line, for example, tells the assembler to use the *offset* of your PREVIOUS_SECTOR procedure.

```
DW      OFFSET _TEXT:PREVIOUS_SECTOR
```

The calculation of this offset is relative to the start of your code segment _TEXT, which is why we put the _TEXT: in front of the procedure name. (As it turns out here, this _TEXT: isn't absolutely necessary. Still, in the interest of clarity, we will write OFFSET _TEXT: anyway.)

Notice that DISPATCH_TABLE contains both byte and word data. This raises a few considerations. In the past, we have always dealt with tables of one type or the other: either all words, or all bytes. But here, we have both, so we have to tell the assembler which type of data to expect when we use a CMP or CALL instruction. In the case of an instruction written as follows, the assembler does not know whether you want to compare words or bytes.

```
CMP     [BX],0
```

By writing the following instruction, you tell the assembler that BX points to a byte, and that you want a byte compare.

```
CMP     BYTE PTR [BX],0
```

Similarly, the instruction CMP WORD PTR [BX],0 would compare words. On the other hand, an instruction like CMP AL,[BX] does not cause problems because AL is a byte register and the assembler knows without being told that you want a byte compare.

Remember that a CALL instruction can be either a NEAR or a FAR CALL. A NEAR CALL needs one word for the address, while the FAR CALL needs two. The following instruction tells the assembler, with WORD PTR, that [BX] points to one word, so it should generate a NEAR CALL and use the word pointed to by [BX] as the address. The address is the one we stored in DISPATCH_TABLE.

```
CALL    WORD PTR [BX]
```

For a FAR CALL, which uses a two-word address, you would use the instruction CALL DWORD PTR [BX]. DWORD stands for *Double Word*, or two words.

As you will see in Chapter 22, you can easily add more key commands to Dskpatch simply by adding more procedures and placing new entries in DISPATCH_TABLE. Right now, you still need to add four procedures before you can test this new version of Dskpatch. You are missing READ_BYTE, WRITE_PROMPT_LINE, PREVIOUS_SECTOR, and NEXT_SECTOR.

READ_BYTE is the procedure needed to read characters and extended ASCII codes from the keyboard. The final version will be able to read special keys (such as the function and cursor keys), ASCII characters, and two-digit hex numbers. At this point, you will write a simple version of READ_BYTE—to read either a character or a special key. The first version of KBD_IO.ASM, which is the file where you will store all of your procedures to read from the keyboard, is as follows:

Listing 19-4 The New File KBD_IO.ASM (Complete Listing in KBD_IO19.ASM)

```
.MODEL  SMALL
.CODE

        PUBLIC  READ_BYTE
;-----------------------------------------------------------------;
; This procedure reads a single ASCII character. This is just      ;
; a test version of READ_BYTE.                                     ;
;                                                                  ;
; Returns:      AL      Character code (unless AH = 1)             ;
;               AH      0 if read ASCII char                       ;
;                       1 if read a special key                    ;
;-----------------------------------------------------------------;
READ_BYTE       PROC
        XOR     AH,AH                   ;Ask for keyboard read function
        INT     16h                     ;Read character/scan code from kbd
        OR      AL,AL                   ;Is it an extended code?
        JZ      EXTENDED_CODE           ;Yes
NOT_EXTENDED:
        XOR     AH,AH                   ;Return just the ASCII code
```

continues

Listing 19-4 continued

```
DONE_READING:
        RET

EXTENDED_CODE:
        MOV     AL,AH                       ;Put scan code into AL
        MOV     AH,1                        ;Signal extended code
        JMP     DONE_READING
READ_BYTE       ENDP


        END
```

READ_BYTE uses a new interrupt, INT 16h, which is an interrupt that gives you access to the keyboard services in the ROM BIOS. Function 0 reads a character from the keyboard without echoing it to the screen. It returns the character code in AL, and the *scan code* in the AH register.

The scan code is the code assigned to each key on the keyboard. Some keys, such as F3, have not been assigned ASCII codes (which means AL will be 0), but they do have scan codes (you will find a table of scan codes in Appendix D). READ_BYTE puts this scan code into the AL register for special keys, and sets AH to 1. Next, add the new procedure WRITE_PROMPT_LINE to DISP_SEC.ASM as follows:

Listing 19-5 Add This Procedure to DISP_SEC.ASM (Complete Listing in DISP_S19.ASM)

```
        PUBLIC  WRITE_PROMPT_LINE
        EXTRN   CLEAR_TO_END_OF_LINE:PROC, WRITE_STRING:PROC
        EXTRN   GOTO_XY:PROC
.DATA
        EXTRN   PROMPT_LINE_NO:BYTE
.CODE
;-------------------------------------------------------------------;
; This procedure writes the prompt line to the screen and clears the  ;
; end of the line.                                                    ;
;                                                                     ;
; On entry:     DS:DX   Address of the prompt-line message            ;
;                                                                          ;
; Uses:         WRITE_STRING, CLEAR_TO_END_OF_LINE, GOTO_XY                ;
; Reads:        PROMPT_LINE_NO                                             ;
;--------------------------------------------------------------------------;
WRITE_PROMPT_LINE       PROC
        PUSH    DX
        XOR     DL,DL                   ;Write the prompt line and
        MOV     DH,PROMPT_LINE_NO       ;move the cursor there
        CALL    GOTO_XY
        POP     DX
        CALL    WRITE_STRING
        CALL    CLEAR_TO_END_OF_LINE
        RET
WRITE_PROMPT_LINE       ENDP
```

There really isn't much to this procedure. It moves the cursor to the beginning of the prompt line, which you set (in DSKPATCH.ASM) to line 21. Then, it writes the prompt line and clears the rest of the line. The cursor is at the end of the prompt when WRITE_PROMPT_LINE is done, and the rest of the line is cleared by CLEAR_TO_END_OF_LINE.

Reading Other Sectors

Finally, you need the two procedures PREVIOUS_SECTOR and NEXT_SECTOR, to read and redisplay the previous and next disk sectors. Add the following two procedures to DISK_IO.ASM.

Listing 19-6 Procedures Added to DISK_IO.ASM (Complete Lsting on DISK_I19.ASM)

```
        PUBLIC  PREVIOUS_SECTOR
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC
.DATA
        EXTRN   CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE
.CODE
```

continues

Listing 19-6 continued

```
;---------------------------------------------------------------------;
; This procedure reads the previous sector, if possible.              ;
;                                                                     ;
; Uses:         WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP              ;
;               WRITE_PROMPT_LINE                                     ;
; Reads:        CURRENT_SECTOR_NO, EDITOR_PROMPT                      ;
; Writes:       CURRENT_SECTOR_NO                                     ;
;---------------------------------------------------------------------;
PREVIOUS_SECTOR         PROC
        PUSH    AX
        PUSH    DX
        MOV     AX,CURRENT_SECTOR_NO    ;Get current sector number
        OR      AX,AX                   ;Don't decrement if already 0
        JZ      DONT_DECREMENT_SECTOR
        DEC     AX
        MOV     CURRENT_SECTOR_NO,AX    ;Save new sector number
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP           ;Display new sector
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
DONT_DECREMENT_SECTOR:
        POP     DX
        POP     AX
        RET
PREVIOUS_SECTOR         ENDP

        PUBLIC  NEXT_SECTOR
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC
.DATA
        EXTRN   CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE
.CODE
;---------------------------------------------------------------------;
; Reads the next sector.                                              ;
;                                                                     ;
; Uses:         WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP              ;
;               WRITE_PROMPT_LINE                                     ;
; Reads:        CURRENT_SECTOR_NO, EDITOR_PROMPT                      ;
; Writes:       CURRENT_SECTOR_NO                                     ;
;---------------------------------------------------------------------;
NEXT_SECTOR     PROC
        PUSH    AX
        PUSH    DX
        MOV     AX,CURRENT_SECTOR_NO
        INC     AX                      ;Move to next sector
        MOV     CURRENT_SECTOR_NO,AX
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP           ;Display new sector
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        POP     DX
        POP     AX
        RET
NEXT_SECTOR     ENDP
```

Now you are ready to build a new version of Dskpatch by assembling all the files you created or changed—Dskpatch, Video_io, Kbd_io, Dispatch, and Disk_io. If you are not using NMake, remember when you Link that there are now seven files— Dskpatch, Disp_sec, Disk_io, Video_io, Kbd_io, Dispatch, and Cursor.

If you are using NMake, you need to make the following additions to the file Makefile are as follows (the backslash at the end of the first line tells Make you are continuing the list of files onto the next line):

Listing 19-7 Changes to the Make File MAKEFILE

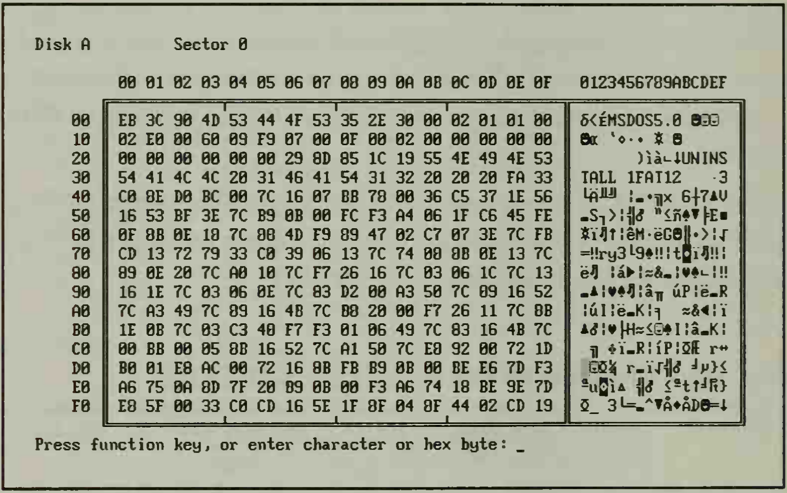
```
dskpatch.exe:   dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
                dispatch.obj kbd_io.obj
        link dskpatch disk_io disp_sec video_io cursor dispatch kbd_io;
                .
                .
                .
cursor.obj:     cursor.asm
        ml /c cursor.asm

dispatch.obj:   dispatch.asm
        ml /c dispatch.asm

kbd_io.obj:     kbd_io.asm
        ml /c kbd_io.asm
```

Remember that the first three lines need to be at the end of your file if you are using Make from MASM 5 or earlier. If you don't have Make or NMake, you may wish to write the following short batch file to link and create your EXE file:

```
LINK DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR DISPATCH KBD_IO;
```

As you add more files, you will only need to change this batch file, rather than type this long link list each time you rebuild the EXE program.

This version of Dskpatch has three active keys: F3 reads and displays the previous sector, stopping at sector 0; F4 reads the next sector; F10 exits from Dskpatch. Give these keys a try. Your display should now look something like Figure 19-1.



```
Disk A        Sector 0

      00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF

  00  EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00   δ<ÉMSDOS5.0 ☻☺☺
  10  02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00   ☻α `○·• ☼ ☻
  20  00 00 00 00 00 00 29 8D 85 1C 19 55 4E 49 4E 53         )ìà∟↓UNINS
  30  54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33   TALL 1FAT12   ·3
  40  C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56   └Ä╨╝ ¦▬•╗x 6┼7▲V
  50  16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE   ▬S┐>¦╣♂ ⁿ≤ñ♠▼╞E■
  60  0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB   ☼ï♫↑¦êM·ëG☻╟•>¦√
  70  CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C   ═‼ry3└9♠‼¦t◘ï♫‼¦
  80  89 0E 20 7C A0 18 7C F7 26 16 7C 03 06 1C 7C 13   ë♫ ¦á↑¦≈&▬¦♥♠∟¦‼
  90  16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52   ▬▲¦♥♠♫¦â╥ úP¦ë▬R
  A0  7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B   ¦úI¦ë▬K¦╕  ≈&◄¦ï
  B0  1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C   ▲♂¦♥├H≈≤☺♠I¦â▬K¦
  C0  00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D    ╗ ♣ï▬R¦íP¦ΦÆ r↔
  D0  B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3   ░☺Φ¼ r▬ï√╣♂ ╛µ}≤
  E0  A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D   ªu◙ì⌂ ╣♂ ≤ªt↑╛₧}
  F0  E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19   Φ_ 3└═▬^▼Å♦ÅD☻═↓

Press function key, or enter character or hex byte: _
```

Figure 19-1: Dskpatch with the prompt line.

Philosophy of the Following Chapters

We covered far more ground than usual in this chapter, and in that respect you have had a taste of the philosophy we will be following in Chapters 20 through 27. From now on, we will clip along at a fairly rapid pace so we can get through more examples of how to write large programs. You will also find more procedures that you can use in your own programs.

These chapters are here for you to learn from, hence the rather high density of new procedures. But in the chapters in Part IV of this book, we'll come back to learning new subjects, so hang on, or (if you wish) skip the remaining chapters on Dskpatch until you're ready to write your own programs. When you are ready to come back again, you will find many useful tidbits for programming.

Of course, if you are chomping at the bit and eager to write your own procedures, read the next chapter. Chapter 20 will give you a number of hints you can use if you want to try writing the procedures in the following chapters.

From Chapter 21 on, we will present many different procedures and let you discover how they work. Why? There are two reasons, both related to setting you on your feet and on your way to assembly language programming. First, we want you to have a library of procedures you can use in your own programs; to use them comfortably, you need to exercise your own skills. Second, by presenting this large programming example, we want not only to show you how to write a large program, but also to give you a feel for it as well.

So take the rest of this book in the way that suits you best. Chapter 20 is for those of you eager to write your own programs. In Chapter 21, we will return to Dskpatch and build the procedures to write and move what we call a phantom cursor: a reverse-video cursor for the hex and ASCII displays.

C H A P T E R

A Programming Challenge

In this chapter we will present a plan for the rest of Dskpatch, which covers Chapters 21 through 27. If you want to try writing some procedures yourself, read this chapter first. You can then read the other chapters if you get stuck, or if you want some hints on how to proceed.

Files altered: None

20

Topics Covered

The Phantom Cursors

Simple Editing

Other Additions and Changes to Dskpatch

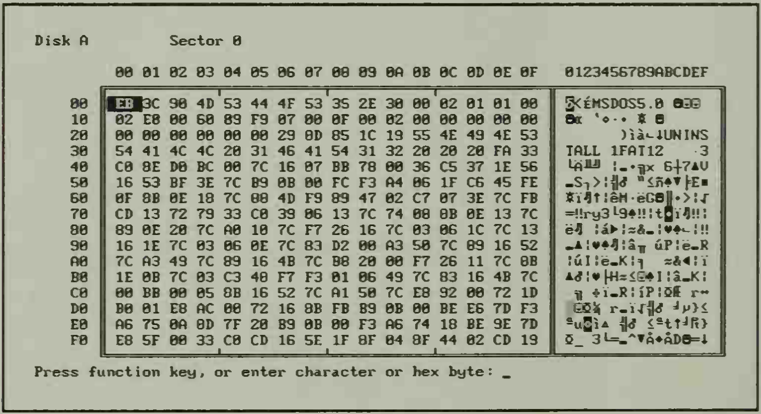
This book contains six more chapters of procedures. If you want to try navigating on your own, read this chapter. We will chart a course for you here, and plot your way through Chapters 21 and 22. Then you can try to write the procedures in each chapter before you read it. If you don't wish to try writing pieces of Dskpatch just yet, skip this chapter for now. It is very brief and leaves many details to your imagination.

You may want to make a copy of all your files before you start making changes. Then when you get to Chapter 21, you will have the choice of following along with the changes, or using your own version.

If you decide to read through this chapter, here is a suggestion on how to proceed—read one section and then try to make your own corresponding changes to Dskpatch. When you feel you have made enough progress, read the chapter with the same name as the section title. After you have read the corresponding chapter, then go on to read the next section.

The Phantom Cursors

In Chapter 21 we will place two phantom cursors on the screen: one in the hex window, and one in the ASCII window. A phantom cursor is similar to a normal cursor, but it does not blink and the background turns white, with the characters black, as displayed in Figure 20-1. The phantom cursor in the hex window is four characters wide; the one in the ASCII window is only one character wide.



```
Disk A          Sector 0

         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F    0123456789ABCDEF

    00   EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00    ë<ÉMSDOS5.0 ☻☺☺
    10   02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00    ☻α `○·• ☼ ☻
    20   00 00 00 00 00 00 29 0D 85 1C 19 55 4E 49 4E 53          )ìà∟↓UNINS
    30   54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33    TALL 1FAT12   ·3
    40   C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56    └Ä╨╝ |▬•╗x 6┼7▲V
    50   16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE    ▬S┐>|╣♂ ⁿ≤ñ♠▼╞E■
    60   0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB    ☼ï♫↑|êM·ëG☻╟•>|√
    70   CD 13 72 79 33 C0 39 06 13 7C 74 08 0B 0E 13 7C    ═‼ry3└9♠‼|t◘♂♫‼|
    80   89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13    ë♫ |á►|≈&▬|♥♠∟|‼
    90   16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52    ▬▲|♥♠♫|â╥ úP|ë▬R
    A0   7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B    |úI|ë▬K|╕  ≈&◄|ï
    B0   1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C    ▲♂|♥├H≈≤☺♠I|â▬K|
    C0   00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D     ╗ ♣ï▬R|íP|ΦÆ r↔
    D0   B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3    ░☺Φ¼ r▬ï√╣♂ ╛µ}≤
    E0   A6 75 0A 0D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D    ªu◙♪⌂ ╣♂ ≤ªt↑╛₧}
    F0   E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19    Φ_ 3└═▬^▼Å♦ÅD☻═↓

 Press function key, or enter character or hex byte: _
```

Figure 20-1: A display with phantom cursors.

How do we create a phantom cursor? Each character on the screen has an *attribute* byte, which you learned about in Chapter 18. This byte tells your PC how to display each character. An attribute code of 7h displays a normal character, while 70h displays a character in inverse video. The latter is exactly what you want for the phantom cursor, so how can you change the attribute of your characters to 70h?

INT 10h function 9 writes both a character and an attribute to the screen; 10h function 8 reads the character code at the current cursor position. You can create a phantom cursor in the hex window with the following steps:

- Save the position of the real cursor (use INT 10h function 3 to read the cursor position and save this in variables).
- Move the real cursor to the start of the phantom cursor in the hex window.
- For the next four characters, read the character code (function 8) and write both the character and its attribute (setting the attribute to 70h).
- Finally, restore the old cursor position.

You can write a phantom cursor in the ASCII window in much the same way. Once you have a working phantom cursor in the hex window, you can add the extra code for the ASCII window.

Keep in mind that your first try is only temporary. Once you have a working program with phantom cursors, you can go back and rewrite your changes, so you have a number of small procedures to do the work. Look at the procedures in Chapter 21 when you are finished to see one way of doing this.

Simple Editing

Once you have your phantom cursors, you will want to move them around on the screen. You need to pay attention to boundary conditions in order to keep the phantom cursors inside each of the two windows. You also want your two phantom cursors to move together because they represent the hex and ASCII representations of the same thing.

How can you move each phantom cursor? Each of the four cursor keys on the keypad sends out a special function number—72 for cursor up; 80 for cursor down; 75 for cursor left; and 77 for cursor right. These are the numbers you need to add to DISPATCH_TABLE, along with the addresses of the four procedures to move the phantom cursors in each of these four directions.

To actually move each phantom cursor, erase it, then change its two coordinates and write it again. If you have been careful about how you wrote the phantom cursors, the four procedures to move them should be fairly simple.

Whenever you type a character on the keyboard, Dskpatch should read this character and replace the byte under the phantom cursor with the character just read. The steps for the simple editing of characters is as follows:

- Read a character from the keyboard.
- Change the hex number in the hex window and the character in the ASCII window to match the character just read.
- Change the byte in the sector buffer, SECTOR.

Here is a simple hint—you don't have to make many changes to add editing. Dispatch requires little more than calling a new procedure (we have called it EDIT_BYTE) that does most of the work. EDIT_BYTE is responsible for changing both the screen and SECTOR.

Other Additions and Changes to Dskpatch

From Chapters 23 through 27, the changes start to become somewhat trickier and more involved. If you are still interested in writing your own version, consider this—what more would you like to see Dskpatch do than it does right now? We have used the following ideas in the remaining chapters.

You will need a new version of READ_BYTE that will read either one character or a two-digit hex number and wait for you to press the Enter key before it returns a character to Dispatch. This part of our "wish list" is not as simple as it sounds. We will spend Chapters 23 and 24 working on this problem.

In Chapter 25, we will go bug hunting; then in Chapter 26 we will learn how to write modified sectors back to the disk using the DOS INT 26h function, which is analogous to the INT 25h that you used to read a sector from the disk. (In Chapter 26 we won't check for read errors, but you will find such checks in the disk version of Dskpatch that is included with this book.)

Finally, in Chapter 27, we will make some changes to Dskpatch so you can see the other half of the sector display. These changes won't allow you to scroll through the sector display as freely as we would like, but those changes are on the disk version of Dskpatch.

C H A P T E R

The Phantom Cursors

In this chapter you will learn how to draw two phantom (or bar) cursors on the screen—one in the hex window, and one in the ASCII window.

Files altered: DISP_SEC.ASM, PHANTOM.ASM, VIDEO_IO.ASM

Disk files: DISP_S21.ASM, PHANTO21.ASM, VIDEO_I21.ASM

21

Topics Covered

The Phantom Cursors

Changing Character Attributes

Summary

In this chapter you will build the procedures to write and erase a phantom cursor in the hex window and another in the ASCII window. A phantom cursor is a shadow that inverts a character, turning the background to white and the character to black. The hex window has the room to make this cursor four characters wide so it will be easy to read. In the ASCII window, the phantom cursor will be one character wide, because there is no room between characters.

The Phantom Cursors

INIT_SEC_DISP is the only procedure that changes the sector display. A new display appears when you start Dskpatch and each time you read a new sector. Since the phantom cursors will be in the sector display, you will begin your work by placing a call to WRITE_PHANTOM in INIT_SEC_DISP. That way, you will write the phantom cursors every time you write a new sector display.

The revised—and final—version of INIT_SEC_DISP in DISP_SEC.ASM is as follows:

Listing 21-1 Changes to INIT_SEC_DISP in DISP_SEC.ASM (Complete listing in DISP_S21.ASM)

```
        PUBLIC  INIT_SEC_DISP
        EXTRN   WRITE_PATTERN:PROC, SEND_CRLF:PROC
        EXTRN   GOTO_XY:PROC, WRITE_PHANTOM:PROC
.DATA
        EXTRN   LINES_BEFORE_SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
.CODE
;-----------------------------------------------------------------------;
;  This procedure initializes the half-sector display.                  ;
;                                                                       ;
; Uses:         WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR              ;
;               WRITE_TOP_HEX_NUMBERS, GOTO_XY, WRITE_PHANTOM           ;
; Reads:        TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN                   ;
;               LINES_BEFORE_SECTOR                                     ;
; Writes:       SECTOR_OFFSET                                           ;
```

```
;-----------------------------------------------------------------;
INIT_SEC_DISP   PROC
        PUSH    DX
        XOR     DL,DL                   ;Move cursor into position
        MOV     DH,LINES_BEFORE_SECTOR
        CALL    GOTO_XY
        CALL    WRITE_TOP_HEX_NUMBERS
        LEA     DX,TOP_LINE_PATTERN
        CALL    WRITE_PATTERN
        CALL    SEND_CRLF
        XOR     DX,DX                   ;Start at the beginning of the sector
        MOV     SECTOR_OFFSET,DX        ;Set sector offset to 0
        CALL    DISP_HALF_SECTOR
        LEA     DX,BOTTOM_LINE_PATTERN
        CALL    WRITE_PATTERN
        CALL    WRITE_PHANTOM           ;Display the phantom cursor
        POP     DX
        RET
INIT_SEC_DISP   ENDP
```

Notice that we have also updated INIT_SEC_DISP to use and initialize variables. It now sets SECTOR_OFFSET to zero to display the first half of a sector.

Let's move on to WRITE_PHANTOM itself. This will take quite a bit of work. Altogether, you have to write six procedures, including WRITE_PHANTOM. First, you will move the real cursor to the position of the phantom cursor in the hex window and change the attribute of the next four characters to inverse video (attribute 70h). This creates a block of white, four characters wide, with the hex number in black. Then you will do the same in the ASCII window, but for a single character. Finally, move the real cursor back to where it was when you started. The procedures for the phantom cursors will be in PHANTOM.ASM, with the exception of WRITE_ATTRIBUTE_N_TIMES, the procedure that will set the attribute of characters. Enter the following procedures into the file PHANTOM.ASM:

Listing 21-2 The new file PHANTOM.ASM (Complete listing in PHANTO21.ASM)

```
.MODEL  SMALL

.DATA

REAL_CURSOR_X           DB      0
REAL_CURSOR_Y           DB      0
        PUBLIC  PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
PHANTOM_CURSOR_X        DB      0
PHANTOM_CURSOR_Y        DB      0

.CCDE

        PUBLIC  MOV_TO_HEX_POSITION
        EXTRN   GOTO_XY:PROC
.DATA
        EXTRN   LINES_BEFORE_SECTOR:BYTE
.CODE
;-------------------------------------------------------------------------;
; This procedure moves the real cursor to the position of the phantom   ;
; cursor in the hex window.                                               ;
;                                                                         ;
; Uses:         GOTO_XY                                                   ;
; Reads:        LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
;-------------------------------------------------------------------------;
MOV_TO_HEX_POSITION     PROC
        PUSH    AX
        PUSH    CX
        PUSH    DX
        MOV     DH,LINES_BEFORE_SECTOR  ;Find row of phantom (0,0)
        ADD     DH,2                    ;Plus row of hex and horizontal bar
        ADD     DH,PHANTOM_CURSOR_Y     ;DH = row of phantom cursor
        MOV     DL,8                    ;Indent on left side
        MOV     CL,3                    ;Each column uses 3 characters, so
        MOV     AL,PHANTOM_CURSOR_X     ; we must multiply CURSOR_X by 3
        MUL     CL
        ADD     DL,AL                   ;And add to the indent, to get column
        CALL    GOTO_XY                 ; for phantom cursor
        POP     DX
        POP     CX
        POP     AX
        RET
MOV_TO_HEX_POSITION     ENDP
```

```
        PUBLIC  MOV_TO_ASCII_POSITION
        EXTRN   GOTO_XY:PROC
.DATA
        EXTRN   LINES_BEFORE_SECTOR:BYTE
.CODE
;-------------------------------------------------------------------;
; This procedure moves the real cursor to the beginning of the phantom  ;
; cursor in the ASCII window.                                       ;
;                                                                   ;
; Uses:         GOTO_XY                                             ;
; Reads:        LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y ;
;-------------------------------------------------------------------;
MOV_TO_ASCII_POSITION   PROC
        PUSH    AX
        PUSH    DX
        MOV     DH,LINES_BEFORE_SECTOR  ;Find row of phantom (0,0)
        ADD     DH,2                    ;Plus row of hex and horizontal bar
        ADD     DH,PHANTOM_CURSOR_Y     ;DH = row of phantom cursor
        MOV     DL,59                   ;Indent on left side
        ADD     DL,PHANTOM_CURSOR_X     ;Add CURSOR_X to get X position
        CALL    GOTO_XY                 ; for phantom cursor
        POP     DX
        POP     AX
        RET
MOV_TO_ASCII_POSITION   ENDP

        PUBLIC  SAVE_REAL_CURSOR
;-------------------------------------------------------------------;
; This procedure saves the position of the real cursor in the two       ;
; variables REAL_CURSOR_X and REAL_CURSOR_Y.                        ;
;                                                                   ;
; Writes:       REAL_CURSOR_X, REAL_CURSOR_Y                        ;
;-------------------------------------------------------------------;
SAVE_REAL_CURSOR        PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Read cursor position
        XOR     BH,BH                   ; on page 0
        INT     10h                     ;And return in DL,DH
        MOV     REAL_CURSOR_Y,DL        ;Save position
        MOV     REAL_CURSOR_X,DH
        POP     DX
```

continues

Listing 21-2 continued

```
        POP     CX
        POP     BX
        POP     AX
        RET
SAVE_REAL_CURSOR        ENDP

        PUBLIC  RESTORE_REAL_CURSOR
        EXTRN   GOTO_XY:PROC
;-----------------------------------------------------------------------;
; This procedure restores the real cursor to its old position, saved in ;
; REAL_CURSOR_X and REAL_CURSOR_Y.                                      ;
;                                                                       ;
; Uses:         GOTO_XY                                                 ;
; Reads:        REAL_CURSOR_X, REAL_CURSOR_Y                            ;
;-----------------------------------------------------------------------;
RESTORE_REAL_CURSOR     PROC
        PUSH    DX
        MOV     DL,REAL_CURSOR_Y
        MOV     DH,REAL_CURSOR_X
        CALL    GOTO_XY
        POP     DX
        RET
RESTORE_REAL_CURSOR     ENDP

        PUBLIC  WRITE_PHANTOM
        EXTRN   WRITE_ATTRIBUTE_N_TIMES:PROC
;-----------------------------------------------------------------------;
; This procedure uses CURSOR_X and CURSOR_Y, through MOV_TO_..., as the ;
; coordinates for the phantom cursor.  WRITE_PHANTOM writes this        ;
; phantom cursor.                                                       ;
;                                                                       ;
; Uses:         WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR               ;
;               RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION                ;
;               MOV_TO_ASCII_POSITION                                   ;
;-----------------------------------------------------------------------;
WRITE_PHANTOM   PROC
        PUSH    CX
        PUSH    DX
        CALL    SAVE_REAL_CURSOR
        CALL    MOV_TO_HEX_POSITION     ;Coord. of cursor in hex window
        MOV     CX,4                    ;Make phantom cursor four chars wide
        MOV     DL,70h
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    MOV_TO_ASCII_POSITION   ;Coord. of cursor in ASCII window
        MOV     CX,1                    ;Cursor is one character wide here
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    RESTORE_REAL_CURSOR
        POP     DX
        POP     CX
        RET
WRITE_PHANTOM   ENDP

        PUBLIC  ERASE_PHANTOM
        EXTRN   WRITE_ATTRIBUTE_N_TIMES:PROC
;-----------------------------------------------------------------------;
; This procedure erases the phantom cursor, just the opposite of        ;
; WRITE_PHANTOM.                                                        ;
;                                                                       ;
; Uses:         WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR               ;
;               RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION                ;
;               MOV_TO_ASCII_POSITION                                   ;
;-----------------------------------------------------------------------;
ERASE_PHANTOM   PROC
        PUSH    CX
        PUSH    DX
        CALL    SAVE_REAL_CURSOR
        CALL    MOV_TO_HEX_POSITION     ;Coord. of cursor in hex window
        MOV     CX,4                    ;Change back to white on black
        MOV     DL,7
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    MOV_TO_ASCII_POSITION
        MOV     CX,1
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    RESTORE_REAL_CURSOR
        POP     DX
        POP     CX
        RET
ERASE_PHANTOM   ENDP


        END
```

WRITE_PHANTOM and ERASE_PHANTOM are much the same. In fact, the only difference is in the attribute used: WRITE_PHANTOM sets the attribute to 70h for inverse video, while ERASE_PHANTOM sets the attribute back to the normal attribute (7). Both of these procedures save the old position of the real cursor with SAVE_REAL_CURSOR. This uses the INT 10h function number 3 to read the position of the cursor and then saves this position in the two bytes REAL_CURSOR_X and REAL_CURSOR_Y.

After saving the real cursor position, both WRITE_PHANTOM and ERASE_PHANTOM then call MOV_TO_HEX_POSITION, which moves the cursor to the start of the phantom cursor in the hex window. Next, WRITE_ATTRIBUTE_N_TIMES writes the inverse-video attribute for four characters, starting at the cursor and moving to the right. This writes the phantom cursor in the hex window. In much the same way, WRITE_PHANTOM then writes a phantom cursor one character wide in the ASCII window. Finally, RESTORE_REAL_CURSOR restores the position of the real cursor to the position it was in before the call to WRITE_PHANTOM. The only procedure left unwritten is WRITE_ATTRIBUTE_N_TIMES, so let's take care of it now.

Changing Character Attributes

You are going to use WRITE_ATTRIBUTE_N_TIMES to do three things. First, it will read the character under the cursor position. You will do this because the INT 10h function you use to set a character's attribute, function number 9, writes both the character *and* the attribute under the cursor. Thus, WRITE_ATTRIBUTE_N_TIMES will change the attribute by writing the new attribute along with the character just read. Finally, the procedure will move the cursor right to the next character position, so you can repeat the whole process N times. You can see the details in the procedure itself. Place WRITE_ATTRIBUTE_N_TIMES in the file VIDEO_IO.ASM as follows:

Listing 21-3 Procedure added to VIDEO_IO.ASM (Complete listing in VIDEO_I21.ASM)

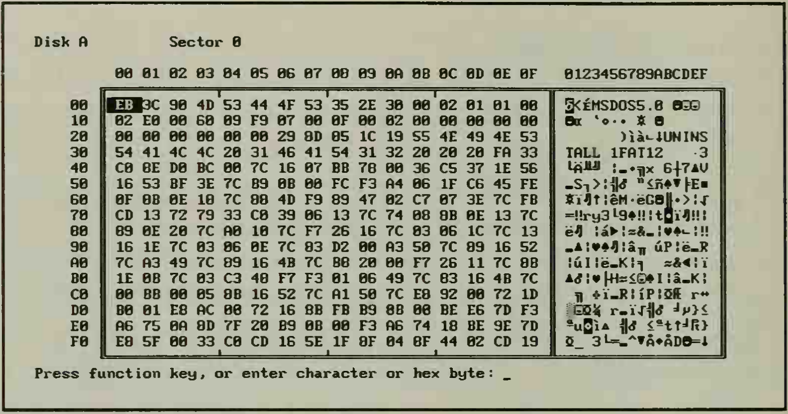
```
        PUBLIC  WRITE_ATTRIBUTE_N_TIMES
        EXTRN   CURSOR_RIGHT:PROC
;----------------------------------------------------------------------;
; This procedure sets the attribute for N characters, starting at the   ;
; current cursor position.                                              ;
;                                                                     ;
; On entry:     CX      Number of characters to set attribute for     ;
;               DL      New attribute for characters                  ;
;                                                                     ;
; Uses:         CURSOR_RIGHT                                          ;
;---------------------------------------------------------------------;
WRITE_ATTRIBUTE_N_TIMES PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BL,DL                   ;Set attribute to new attribute
        XOR     BH,BH                   ;Set display page to 0
        MOV     DX,CX                   ;CX is used by the BIOS routines
        MOV     CX,1                    ;Set attribute for one character
ATTR_LOOP:
        MOV     AH,8                    ;Read character under cursor
        INT     10h
        MOV     AH,9                    ;Write attribute/character
        INT     10h
        CALL    CURSOR_RIGHT
        DEC     DX                      ;Set attribute for N characters?
        JNZ     ATTR_LOOP               ;No, continue
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
WRITE_ATTRIBUTE_N_TIMES ENDP
```

This is both the first and final version of WRITE_ATTRIBUTE_N_TIMES. With it, you have also created the final version of VIDEO_IO.ASM, so you won't need to change or assemble it again.

Summary

You now have eight files to link, with the main procedure still in Dskpatch. Of these, you have changed two files, Disp_sec and Video_io, and created one, Phantom. If you are using NMake or the short batch file we suggested in Chapter 20, remember to add your new file, Phantom, to the list.

When you run Dskpatch now, you will see it write the sector display, just as before, but Dskpatch will also write in the two phantom cursors in Figure 21-1. Notice that the real cursor is back where it should be at the very end.



```
 Disk A          Sector 0

             00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F   0123456789ABCDEF

        00   EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 01 01 00   ΣKÉMSDOS5.0 ☻☺☺
        10   02 E0 00 60 09 F9 07 00 0F 00 02 00 00 00 00 00   ☻α `○·• ☼ ☻
        20   00 00 00 00 00 00 29 8D 05 1C 19 55 4E 49 4E 53         )ìà∟↓UNINS
        30   54 41 4C 4C 20 31 46 41 54 31 32 20 20 20 FA 33   TALL 1FAT12   ·3
        40   C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56   └Ä╨╝ ¦▬•╗x 6┼7▲V
        50   16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE   ▬S┐>¦╣♂ ⁿ≤ñ♠▼╞E■
        60   0F 8B 0E 10 7C 88 4D F9 89 47 02 C7 07 3E 7C FB   ☼ï♫►¦êM·ëG☻╟•>¦√
        70   CD 13 72 79 33 C0 39 06 13 7C 74 08 8B 0E 13 7C   ═‼ry3└9♠‼¦t◘ï♫‼¦
        80   89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13   ë♫ ¦á►¦≈&▬¦♥♠∟¦‼
        90   16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52   ▬▲¦♥♠♫¦â╥ úP¦ë▬R
        A0   7C A3 49 7C 89 16 4B 7C B8 20 00 F7 26 11 7C 8B   ¦úI¦ë▬K¦╕  ≈&◄¦ï
        B0   1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C   ▲♂¦♥├H≈≤☺♠I¦â▬K¦
        C0   00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D    ╗ ♣ï▬R¦íP¦Φ Æ r↔
        D0   B0 01 E8 AC 00 72 16 8B FB B9 0B 00 BE E6 7D F3   ░☺Φ¼ r▬ï√╣♂ ╛µ}≤
        E0   A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D   ªu◙ì⌂ ╣♂ ≤ªt↑╛₧}
        F0   E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19   Φ_ 3└═▬^▼Å♦ÅD☻═↓

 Press function key, or enter character or hex byte: _
```

Figure 21-1: Screen display with phantom cursors.

The next chapter covers adding procedures to move your newly formed phantom cursors. It will also cover how to add a simple editing procedure which allows you to change the byte under the phantom cursor.

CHAPTER

Simple Editing

In this chapter you will add routines that allow you to move the phantom cursor around on your screen. Then you will add a simple procedure that allows you to input new values into the sector display.

Files altered: DISPATCH.ASM, EDITOR.ASM, PHANTOM.ASM

Disk files: DISPAT22.ASM, EDITOR22.ASM, PHANTO22.ASM

22

Topics Covered

You have almost reached the point at which you can begin to edit your sector display—change numbers in the half sector display. You will soon add simple versions of the procedures for editing bytes in the display, but before you do, you need some way to move the phantom cursors to different bytes within the half-sector display. This task turns out to be fairly simple, now that you have the two procedures ERASE_PHANTOM and WRITE_PHANTOM.

Moving the Phantom Cursors

Moving the phantom cursors in any direction depends on three basic steps: erasing the phantom cursor at its current position; changing the cursor position by changing one of the variables, PHANTOM_CURSOR_X or PHANTOM_CURSOR_Y; and using WRITE_PHANTOM to write the phantom cursor at the new position. You must be careful not to let the cursor move outside the window, which is 16 bytes wide and 16 bytes high.

To move the phantom cursors, you will need four new procedures, one for each of the arrow keys on the keyboard. DISPATCHER needs no changes because the information on procedures and extended codes is in the table DISPATCH_TABLE. You just need to add the extended ASCII codes and addresses of the procedures for each of the arrow keys. Additions to DISPATCH.ASM that will bring the cursor keys to life are as follows:

Listing 22-1 Changes to DISPATCH.ASM

```
.MODEL  SMALL

.CODE
        EXTRN   NEXT_SECTOR:PROC                        ;In DISK_IO.ASM
        EXTRN   PREVIOUS_SECTOR:PROC                    ;In DISK_IO.ASM
        EXTRN   PHANTOM_UP:PROC, PHANTOM_DOWN:PROC      ;In PHANTOM.ASM
        EXTRN   PHANTOM_LEFT:PROC, PHANTOM_RIGHT:PROC
.DATA
```

```
;----------------------------------------------------------------------;
; This table contains the legal extended ASCII keys and the addresses  ;
; of the procedures that should be called when each key is pressed.   ;
;                                                                      ;
; The format of the table is                                           ;
;               DB      72              ;Extended code for cursor up   ;
;               DW      OFFSET _TEXT:PHANTOM_UP                        ;
;----------------------------------------------------------------------;
DISPATCH_TABLE  LABEL   BYTE
        DB      61                              ;F3
        DW      OFFSET _TEXT:PREVIOUS_SECTOR
        DB      62                              ;F4
        DW      OFFSET _TEXT:NEXT_SECTOR
        DB      72                              ;Cursor up
        DW      OFFSET _TEXT:PHANTOM_UP
        DB      80                              ;Cursor down
        DW      OFFSET _TEXT:PHANTOM_DOWN
        DB      75                              ;Cursor left
        DW      OFFSET _TEXT:PHANTOM_LEFT
        DB      77                              ;Cursor right
        DW      OFFSET _TEXT:PHANTOM_RIGHT
        DB      0                               ;End of the table
```

As you can see, it is simple to add commands to Dskpatch by placing the procedure names in DISPATCH_TABLE and writing the procedures.

The procedures PHANTOM_UP, PHANTOM_DOWN, and so on are fairly simple. They are also quite similar to one another, differing only in the boundary conditions used for each. We have already described how they work; see if you can write them yourself, in the file PHANTOM.ASM, before you read on. Our versions of the procedures to move the phantom cursors are as follows:

Listing 22-2 Add these procedures to PHANTOM.ASM (Complete listing in PHANTO22.ASM)

```
;-----------------------------------------------------------------------;
; These four procedures move the phantom cursors.                       ;
;                                                                       ;
; Uses:           ERASE_PHANTOM, WRITE_PHANTOM                          ;
; Reads:          PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y                    ;
; Writes:         PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y                    ;
;-----------------------------------------------------------------------;

        PUBLIC  PHANTOM_UP
PHANTOM_UP      PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        DEC     PHANTOM_CURSOR_Y        ;Move cursor up one line
        JNS     WASNT_AT_TOP            ;Was not at the top, write cursor
        MOV     PHANTOM_CURSOR_Y,0      ;Was at the top, so put back there
WASNT_AT_TOP:
        CALL    WRITE_PHANTOM           ;Write the phantom at new position
        RET
PHANTOM_UP      ENDP

        PUBLIC  PHANTOM_DOWN
PHANTOM_DOWN    PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        INC     PHANTOM_CURSOR_Y        ;Move cursor down one line
        CMP     PHANTOM_CURSOR_Y,16     ;Was it at the bottom?
        JB      WASNT_AT_BOTTOM         ;No, so write phantom
        MOV     PHANTOM_CURSOR_Y,15     ;Was at bottom, so put back there
WASNT_AT_BOTTOM:
        CALL    WRITE_PHANTOM           ;Write the phantom cursor
        RET
PHANTOM_DOWN    ENDP

        PUBLIC  PHANTOM_LEFT
PHANTOM_LEFT    PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        DEC     PHANTOM_CURSOR_X        ;Move cursor left one column
        JNS     WASNT_AT_LEFT           ;Was not at the left side, write cursor
        MOV     PHANTOM_CURSOR_X,0      ;Was at left, so put back there
```

```
WASNT_AT_LEFT:
        CALL    WRITE_PHANTOM               ;Write the phantom cursor
        RET
PHANTOM_LEFT    ENDP

        PUBLIC  PHANTOM_RIGHT
PHANTOM_RIGHT   PROC
        CALL    ERASE_PHANTOM               ;Erase at current position
        INC     PHANTOM_CURSOR_X            ;Move cursor right one column
        CMP     PHANTOM_CURSOR_X,16         ;Was it already at the right side?
        JB      WASNT_AT_RIGHT
        MOV     PHANTOM_CURSOR_X,15         ;Was at right, so put back there
WASNT_AT_RIGHT:
        CALL    WRITE_PHANTOM               ;Write the phantom cursor
        RET
PHANTOM_RIGHT   ENDP
```

PHANTOM_LEFT and PHANTOM_RIGHT are the final versions, but you will have to change PHANTOM_UP and PHANTOM_DOWN when you begin to scroll the display.

Test Dskpatch now to see if you can move the phantom cursors around on the screen. They should move together, and stay within their own windows.

As Dskpatch stands now, you can see only the first half of a sector. In Chapter 27, you will make some additions and changes to Dskpatch so you can scroll the display to see other parts of the sector. At that time, you will change both PHANTOM_UP and PHANTOM_DOWN to scroll the screen when you try to move the cursor beyond the top or bottom of the screen. For example, when the cursor is at the bottom of the half-sector display, pushing the cursor-down key again should scroll the display up one line, adding another line at the bottom, so that you see the next 16 bytes. Scrolling is rather messy, however, so we will save these procedures until almost the end. Through Chapter 26, we will develop the editing and keyboard-input sections of Dskpatch by using only the first half sector. Now, let's go on to add editing, so you can change bytes on your display.

Simple Editing

You already have a simple keyboard-input procedure, READ_BYTE, which reads one character from the keyboard without waiting for you to press the Enter key. You will use this old test version of READ_BYTE to develop editing. In the next chapter, you will write a more sophisticated version of the procedure that waits until you press either the Enter key or a special key, such as a function or cursor key.

The editing procedure will be called EDIT_BYTE. It will change one byte both on the screen and in memory (SECTOR). EDIT_BYTE will take the character in the DL register, write it to the memory location within SECTOR that is currently pointed to by the phantom cursor, and then change the display.

DISPATCHER already has a nice niche where you can place a CALL to EDIT_BYTE. The new version of DISPATCHER in DISPATCH.ASM, with the CALL to EDIT_BYTE and the changes to go along with it, is as follows:

Listing 22-3 Changes to DISPATCHER in DISPATCH.ASM (Complete listing in DISPAT22.ASM)

```
        PUBLIC  DISPATCHER
        EXTRN   READ_BYTE:PROC, EDIT_BYTE:PROC
;---------------------------------------------------------------------;
; This is the central dispatcher.  During normal editing and viewing,  ;
; this procedure reads characters from the keyboard and, if the character;
; is a command key (such as a cursor key), DISPATCHER calls the        ;
; procedures that do the actual work.  This dispatching is done for    ;
; special keys listed in the table DISPATCH_TABLE, where the procedure ;
; addresses are stored just after the key names.                      ;
;                                                                      ;
; If the character is not a special key, then it should be placed      ;
; directly into the sector buffer—this is the editing mode.            ;
;                                                                      ;
; Uses:         READ_BYTE, EDIT_BYTE                                   ;
;---------------------------------------------------------------------;
DISPATCHER      PROC
        PUSH    AX
        PUSH    BX
```

```
DISPATCH_LOOP:
        CALL    READ_BYTE               ;Read character into AL
        OR      AH,AH                   ;AX = -1 if no character read, 1
                                        ; for an extended code.
        JS      DISPATCH_LOOP           ;No character read, try again
        JNZ     SPECIAL_KEY             ;Read extended code
~~; do nothing with the character for now~~
        MOV     DL,AL
        CALL    EDIT_BYTE               ;Was normal character, edit byte
        JMP     DISPATCH_LOOP           ;Read another character

SPECIAL_KEY:
        CMP     AL,68                   ;F10—exit?
        JE      END_DISPATCH            ;Yes, leave
                                        ;Use BX to look through table
        LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
        CMP     BYTE PTR [BX],0         ;End of table?
        JE      NOT_IN_TABLE            ;Yes, key was not in the table
        CMP     AL,[BX]                 ;Is it this table entry?
        JE      DISPATCH                ;Yes, then dispatch
        ADD     BX,3                    ;No, try next entry
        JMP     SPECIAL_LOOP            ;Check next table entry

DISPATCH:
        INC     BX                      ;Point to address of procedure
        CALL    WORD PTR [BX]           ;Call procedure
        JMP     DISPATCH_LOOP           ;Wait for another key

NOT_IN_TABLE:                           ;Do nothing, just read next character
        JMP     DISPATCH_LOOP

END_DISPATCH:
        POP     DX
        POP     BX
        POP     AX
        RET
DISPATCHER      ENDP
```

The EDIT_BYTE procedure does a lot of work almost entirely by calling other procedures; this is one feature of modular design. Modular design enables you to write complex procedures by giving a list of CALLs to other procedures that do the work. Many of the procedures in EDIT_BYTE already work with a character in the DL register, so the only instruction other than a CALL (or PUSH, POP) is the LEA instruction to set the address of the prompt for WRITE_PROMPT_LINE. Most of the procedure calls in EDIT_BYTE are for updating the display when you edit a byte. You will see the other details of EDIT_BYTE when you come to the procedure listing.

Because EDIT_BYTE changes the byte on-screen, you need the procedure WRITE_TO_MEMORY to change the byte in SECTOR. WRITE_TO_MEMORY uses the coordinates in PHANTOM_CURSOR_X and PHANTOM_CURSOR_Y to calculate the offset into SECTOR of the phantom cursor. Then it writes the character (byte) in the DL register to the correct byte within SECTOR. The new file, EDITOR.ASM, which contains the final versions of both EDIT_BYTE and WRITE_TO_MEMORY is as follows:

Listing 22-4 The new file EDITOR.ASM (Complete listing in EDITOR22.ASM)

```
.MODEL  SMALL

.CODE

.DATA
        EXTRN   SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   PHANTOM_CURSOR_X:BYTE
        EXTRN   PHANTOM_CURSOR_Y:BYTE
.CODE
;-------------------------------------------------------------------;
; This procedure writes one byte to SECTOR, at the memory location   ;
; pointed to by the phantom cursor.                                  ;
;                                                                    ;
; On entry:     DL      Byte to write to SECTOR                      ;
;                                                                    ;
; The offset is calculated by                                        ;
;   OFFSET = SECTOR_OFFSET + (16 * PHANTOM_CURSOR_Y) + PHANTOM_CURSOR_X ;
;                                                                    ;
; Reads:        PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y, SECTOR_OFFSET    ;
; Writes:       SECTOR                                               ;
;-------------------------------------------------------------------;
```

```
WRITE_TO_MEMORY PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,SECTOR_OFFSET
        MOV     AL,PHANTOM_CURSOR_Y
        XOR     AH,AH
        MOV     CL,4                    ;Multiply PHANTOM_CURSOR_Y by 16
        SHL     AX,CL
        ADD     BX,AX                   ;BX = SECTOR_OFFSET + (16 * Y)
        MOV     AL,PHANTOM_CURSOR_X
        XOR     AH,AH
        ADD     BX,AX                   ;That's the address!
        MOV     SECTOR[BX],DL           ;Now, store the byte
        POP     CX
        POP     BX
        POP     AX
        RET
WRITE_TO_MEMORY ENDP


        PUBLIC  EDIT_BYTE
        EXTRN   SAVE_REAL_CURSOR:PROC, RESTORE_REAL_CURSOR:PROC
        EXTRN   MOV_TO_HEX_POSITION:PROC, MOV_TO_ASCII_POSITION:PROC
        EXTRN   WRITE_PHANTOM:PROC, WRITE_PROMPT_LINE:PROC
        EXTRN   CURSOR_RIGHT:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC
.DATA
        EXTRN   EDITOR_PROMPT:BYTE
.CODE
;-----------------------------------------------------------------------;
; This procedure changes a byte in memory and on the screen.            ;
;                                                                       ;
; On entry:     DL      Byte to write into SECTOR, and change on screen ;
;                                                                       ;
; Uses:         SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR                   ;
;               MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION              ;
;               WRITE_PHANTOM, WRITE_PROMPT_LINE, CURSOR_RIGHT          ;
;               WRITE_HEX, WRITE_CHAR, WRITE_TO_MEMORY                  ;
; Reads:        EDITOR_PROMPT                                           ;
;-----------------------------------------------------------------------;
EDIT_BYTE       PROC
        PUSH    DX
        CALL    SAVE_REAL_CURSOR
        CALL    MOV_TO_HEX_POSITION     ;Move to the hex number in the
        CALL    CURSOR_RIGHT            ; hex window
```

Listing 22-4 continued

```
        CALL    WRITE_HEX               ;Write the new number
        CALL    MOV_TO_ASCII_POSITION   ;Move to the char. in the ASCII
window
        CALL    WRITE_CHAR              ;Write the new character
        CALL    RESTORE_REAL_CURSOR     ;Move cursor back where it belongs
        CALL    WRITE_PHANTOM           ;Rewrite the phantom cursor
        CALL    WRITE_TO_MEMORY         ;Save this new byte in SECTOR
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        POP     DX
        RET
EDIT_BYTE       ENDP


        END
```

Build this new version and see how it works. By the way, if you see a Link error message about an unresolved external, this probably means you forgot to add EDITOR.ASM to your Makefile. As you type letters in this new version, the byte under the phantom cursor should change instantly. The version of EDIT_BYTE you are using here changes a byte as soon as you type any key. This means that you cannot enter hex numbers, you can only type letters and other symbols on your keyboard. We will show you how to fix this shortcoming in the next chapters.

Summary

Dskpatch now consists of nine files: Dskpatch, Dispatch, Disp_sec, Disk_io, Video_io, Kbd_io, Phantom, Cursor, and Editor. In this chapter, you changed Dispatch and added Editor. None of these files is very long, so they do not take long to assemble. Furthermore, you can make changes fairly quickly by editing one of these files, reassembling it, and then linking all the files together again.

In terms of the current version of Dskpatch, push any key and you will see a change in the number and character under the phantom cursor. Editing works, but it is not very safe yet, since you can change a byte by hitting any key. You need to build in some type of safeguard, such as pressing Enter to change a byte, so you don't make an accidental change by leaning on the keyboard unintentionally.

In addition, the current version of READ_BYTE does not allow you to enter a hex number to change a byte. Chapter 24 will cover rewriting READ_BYTE to allow you to enter a two-digit hex number and to accept a new character by pressing the Enter key. Before rewriting READ_BYTE, you will have to write a hex input procedure. In the next chapter, you will write input procedures for both hex and decimal.

C H A P T E R

Hex and Decimal Input

In this chapter you will build two subroutines, READ_BYTE and READ_DECIMAL to read hex and decimal numbers. You will build and test these subroutines using a test program, rather than the full Dskpatch program. This makes it easier to test subroutines to make sure they are working properly.

Files altered: KBD_IO.ASM, TEST.ASM

Disk files: KBD_IO23.ASM, TEST23.ASM

23

Topics Covered

Hex Input

Decimal Input

Summary

You will encounter two new procedures for keyboard input in this chapter: one procedure for reading a byte by reading either a two-digit hex number or a single character, and another for reading a word by reading the characters of a decimal number. These will be the hex and decimal input procedures.

Both procedures are sufficiently tricky that you need to use a test program with them before linking them into Dskpatch. You will be working with READ_BYTE, and a test procedure will be particularly important here because this procedure will (temporarily) lose its ability to read special function keys. Since Dskpatch relies on the function keys, there won't be a way to quit Dskpatch, therefore you will not be able to use the new READ_BYTE with Dskpatch. You will also find out why you cannot read special function keys with the READ_BYTE developed here. In the next chapter, we will modify the file to make function-key problems go away.

Hex Input

Let's begin by rewriting READ_BYTE. In the last chapter, READ_BYTE would read either an ordinary character or a special function key, and return one byte to Dispatch. Dispatch would then call the Editor if READ_BYTE read an ordinary character; EDIT_BYTE would modify the byte pointed to by the phantom cursor. If not, Dispatch looked for special function keys in DISPATCH_TABLE to see if the byte was there; if so, Dispatch called the procedure named in the table.

As mentioned in Chapter 22, the old version of READ_BYTE makes it too easy to change a byte by accident. If you unintentionally hit any key on the keyboard (other than special keys), EDIT_BYTE will change the byte under the phantom cursor. Sometimes people are clumsy, and such an inadvertent change in a sector can lead to disaster.

In this chapter, you will change READ_BYTE so that it won't return the character typed until you press the Enter key. You will add this feature by using the DOS INT 21h function 0Ah to read a string of characters. DOS only returns this string when you press Enter. Along the way, you will lose special function keys, for reasons which become apparent later.

To see exactly how your changes affect READ_BYTE, you need to write a test program to test READ_BYTE in isolation. That way, if anything strange happens you will know it is READ_BYTE and not some other part of Dskpatch. The job of writing a test procedure will be simpler if you use a few procedures from Kbd_io, Video_io, and Cursor to print information on the progress of READ_BYTE. You will use procedures such as WRITE_HEX and WRITE_DECIMAL to print the character code returned and the number of characters read. The details are in TEST.ASM as follows:

Listing 23-1 The test program TEST.ASM

```
.MODEL  SMALL

.STACK

.DATA
ENTER_PROMPT            DB      'Enter characters: ',0
CHARACTER_PROMPT        DB      'Character code: ',0
SPECIAL_CHAR_PROMPT     DB      'Special character read: ',0

.CODE
        EXTRN   WRITE_HEX:PROC, WRITE_DECIMAL:PROC
        EXTRN   WRITE_STRING:PROC, SEND_CRLF:PROC
        EXTRN   READ_BYTE:PROC

TEST_READ_BYTE  PROC
        MOV     AX,DGROUP
        MOV     DS,AX

        LEA     DX,ENTER_PROMPT
        CALL    WRITE_STRING
        CALL    READ_BYTE
        CALL    SEND_CRLF
        LEA     DX,CHARACTER_PROMPT
        CALL    WRITE_STRING
        MOV     DL,AL
        CALL    WRITE_HEX
        CALL    SEND_CRLF
        LEA     DX,SPECIAL_CHAR_PROMPT
```

continues

Listing 23-1 continued

```
        CALL    WRITE_STRING
        MOV     DL,AH
        XOR     DH,DH
        CALL    WRITE_DECIMAL
        CALL    SEND_CRLF

        MOV     AH,4Ch                      ;Return to DOS
        INT     21h
TEST_READ_BYTE  ENDP


        END     TEST_READ_BYTE
```

To assemble this file, link it with your current versions of Kbd_io, Video_io, and Cursor (place Test first in the LINK list), and then run Test. If you press any special function key, Test will display the scan code and a 1 to tell you that you typed a special character. Otherwise, it will display 0 (no special key).

The bulk of the instructions in TEST.ASM are for formatting. One thing you may have noticed is that we have used some of the procedures in Kbd_io, Video_io, and Cursor without regard to the other files in our project. We could do this because we were careful to place only general-purpose procedures into these files. In other words, Kbd_io, Video_io, and Cursor are designed to be used by any program you write. It is a good idea to separate your procedures by source file into general-purpose and specific procedures so you can easily reuse general-purpose procedures in new programs you write.

Let's move on to rewriting READ_BYTE to accept a string of characters. Not only will this save you from your clumsiness when you use Dskpatch, it will also allow you to use the Backspace key to delete characters if you change your mind about what you want to type. READ_BYTE will use the procedure READ_STRING to read a string of characters.

READ_STRING is very simple, almost trivial, but we have placed it in a separate procedure so you can rewrite it in the next chapter to read special function keys without having to press the Enter key. To save time, we will also add three other procedures that READ_BYTE uses: STRING_TO_UPPER, CONVERT_HEX_DIGIT, and HEX_TO_BYTE.

STRING_TO_UPPER and HEX_TO_BYTE both work on strings. STRING_TO_UPPER converts all the lowercase letters in a string to uppercase. That means you can type either f3 or F3 for the hex number F3h. By allowing hex numbers to be typed in either lower- or uppercase letters, we add user-friendliness to Dskpatch.

HEX_TO_BYTE takes the string read by DOS, after you call STRING_TO_UPPER, and converts the two-digit hex string to a single-byte number. HEX_TO_BYTE makes use of CONVERT_HEX_DIGIT to convert each hex digit to a four-bit number.

How do you ensure that DOS won't read more than two hex digits? The DOS function 0Ah reads an entire string of characters into an area of memory defined like this:

```
CHAR_NUM_LIMIT  DB      0
NUM_CHARS_READ  DB      0
STRING          DB      80 DUP (0)
```

The first byte ensures that you don't read too many characters. CHAR_NUM_LIMIT tells DOS how many characters, at most, to read. If you set this to three, DOS will read up to two characters, plus the carriage-return character (DOS always counts the carriage return). Any characters typed after that will be discarded, and DOS will beep to let you know you have passed the limit. When you press the Enter key, DOS sets the second byte, NUM_CHARS_READ, to the number of characters it actually read, not including the carriage return.

READ_BYTE and STRING_TO_UPPER both use NUM_CHARS_READ. For example, READ_BYTE checks NUM_CHARS_READ to find out whether you typed a single character or a two-digit hex number. If NUM_CHARS_READ was set to one, READ_BYTE returns a single character in the AL register. If NUM_CHARS_READ was set to two, READ_BYTE uses HEX_TO_BYTE to convert the two-digit hex string to a byte.

The new file KBD_IO.ASM, with all four new procedures is shown in Listing 23-2. (Notice that the old READ_BYTE was kept by renaming it to READ_KEY. This will be used in the next chapter.)

Listing 23-2 The New Version of KBD_IO.ASM

```
.MODEL  SMALL

.DATA

KEYBOARD_INPUT  LABEL   BYTE
CHAR_NUM_LIMIT  DB      0               ;Length of input buffer
NUM_CHARS_READ  DB      0               ;Number of characters read
CHARS           DB      80 DUP (0)      ;A buffer for keyboard input

.CODE

        PUBLIC  STRING_TO_UPPER
;-------------------------------------------------------------------------;
; This procedure converts the string, using the DOS format for strings, ;
; to all uppercase letters.                                             ;
;                                                                       ;
; On entry:     DS:DX   Address of string buffer                        ;
;-------------------------------------------------------------------------;
STRING_TO_UPPER PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                      ;Point to character count
        MOV     CL,[BX]                 ;Character count in 2nd byte of buffer
        XOR     CH,CH                   ;Clear upper byte of count
UPPER_LOOP:
        INC     BX                      ;Point to next character in buffer
        MOV     AL,[BX]
        CMP     AL,'a'                  ;See if it is a lowercase letter
        JB      NOT_LOWER               ;Nope
        CMP     AL,'z'
        JA      NOT_LOWER
        ADD     AL,'A'-'a'              ;Convert to uppercase letter
        MOV     [BX],AL
NOT_LOWER:
        LOOP    UPPER_LOOP
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_TO_UPPER ENDP

;-----------------------------------------------------------------------;
; This procedure converts a character from ASCII (hex) to a nibble     ;
; (4 bits).                                                             ;
;                                                                       ;
; On entry:     AL      Character to convert                            ;
; Returns:      AL      Nibble                                          ;
;               CF      Set for error, cleared otherwise                ;
;-----------------------------------------------------------------------;
CONVERT_HEX_DIGIT       PROC
        CMP     AL,'0'                  ;Is it a legal digit?
        JB      BAD_DIGIT               ;Nope
        CMP     AL,'9'                  ;Not sure yet
        JA      TRY_HEX                 ;Might be hex digit
        SUB     AL,'0'                  ;Is decimal digit, convert to nibble
        CLC                             ;Clear the carry, no error
        RET
TRY_HEX:
        CMP     AL,'A'                  ;Not sure yet
        JB      BAD_DIGIT               ;Not hex
        CMP     AL,'F'                  ;Not sure yet
        JA      BAD_DIGIT               ;Not hex
        SUB     AL,'A'-10               ;Is hex, convert to nibble
        CLC                             ;Clear the carry, no error
        RET
BAD_DIGIT:
        STC                             ;Set the carry, error
        RET
CONVERT_HEX_DIGIT       ENDP

        PUBLIC  HEX_TO_BYTE
;-----------------------------------------------------------------------;
; This procedure converts the two characters at DS:DX from hex to one   ;
; byte.                                                                 ;
;                                                                       ;
; On entry:     DS:DX   Address of two characters for hex number        ;
; Returns:      AL      Byte                                            ;
;               CF      Set for error, clear if no error                ;
;                                                                       ;
; Uses:         CONVERT_HEX_DIGIT                                       ;
;-----------------------------------------------------------------------;
```

continues

Listing 23-2 continued

```
HEX_TO_BYTE     PROC
        PUSH    BX
        PUSH    CX
        MOV     BX,DX                   ;Put address in BX for indirect addr
        MOV     AL,[BX]                 ;Get first digit
        CALL    CONVERT_HEX_DIGIT
        JC      BAD_HEX                 ;Bad hex digit if carry set
        MOV     CX,4                    ;Now multiply by 16
        SHL     AL,CL
        MOV     AH,AL                   ;Retain a copy
        INC     BX                      ;Get second digit
        MOV     AL,[BX]
        CALL    CONVERT_HEX_DIGIT
        JC      BAD_HEX                 ;Bad hex digit if carry set
        OR      AL,AH                   ;Combine two nibbles
        CLC                             ;Clear carry for no error
DONE_HEX:
        POP     CX
        POP     BX
        RET
BAD_HEX:
        STC                             ;Set carry for error
        JMP     DONE_HEX
HEX_TO_BYTE     ENDP


;-------------------------------------------------------------------;
; This is a simple version of READ_STRING.                          ;
;                                                                   ;
; On entry:     DS:DX   Address of string area                      ;
;-------------------------------------------------------------------;
READ_STRING     PROC
        PUSH    AX
        MOV     AH,0Ah                  ;Call for buffered keyboard input
        INT     21h                     ;Call DOS function for buffered
input
        POP     AX
        RET
READ_STRING     ENDP

        PUBLIC  READ_BYTE
```

```
;----------------------------------------------------------------------;
; This procedure reads either a single ASCII character or a two-digit  ;
; hex number.  This is just a test version of READ_BYTE.               ;
;                                                                      ;
; Returns:      AL      Character code (unless AH = 0)                 ;
;               AH      0 if read ASCII char                           ;
;                       1 if read a special key                        ;
;                       -1 if no characters read                       ;
;                                                                      ;
; Uses:         HEX_TO_BYTE, STRING_TO_UPPER, READ_STRING              ;
; Reads:        KEYBOARD_INPUT, etc.                                   ;
; Writes:       KEYBOARD_INPUT, etc.                                   ;
;----------------------------------------------------------------------;
READ_BYTE       PROC
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,3        ;Allow only two characters (plus
                                         Enter)
        LEA     DX,KEYBOARD_INPUT
        CALL    READ_STRING
        CMP     NUM_CHARS_READ,1        ;See how many characters
        JE      ASCII_INPUT             ;Just one, treat as ASCII character
        JB      NO_CHARACTERS           ;Only Enter key hit
        CALL    STRING_TO_UPPER         ;No, convert string to uppercase
        LEA     DX,CHARS                ;Address of string to convert
        CALL    HEX_TO_BYTE             ;Convert string from hex to byte
        JC      NO_CHARACTERS           ;Error, so return 'no characters
                                         read'
        XOR     AH,AH                   ;Signal read one byte
DONE_READ:
        POP     DX
        RET
NO_CHARACTERS:
        XOR     AH,AH                   ;Set to 'no characters read'
        NOT     AH                      ;Return -1 in AH
        JMP     DONE_READ
ASCII_INPUT:
        MOV     AL,CHARS                ;Load character read
        XOR     AH,AH                   ;Signal read one byte
        JMP     DONE_READ
READ_BYTE       ENDP


        PUBLIC  READ_KEY
```

continues

Listing 23-2 continued

```
;---------------------------------------------------------------------;
; This procedure reads one key from the keyboard.                     ;
;                                                                     ;
; Returns:      AL      Character code (unless AH = 1)                ;
;               AH      0 if read ASCII char                          ;
;                       1 if read a special key                       ;
;---------------------------------------------------------------------;
READ_KEY        PROC
        XOR     AH,AH                   ;Ask for keyboard read function
        INT     16h                     ;Read character/scan code from keyboard
        OR      AL,AL                   ;Is it an extended code?
        JZ      EXTENDED_CODE           ;Yes
NOT_EXTENDED:
        XOR     AH,AH                   ;Return just the ASCII code
DONE_READING:
        RET

EXTENDED_CODE:
        MOV     AL,AH                   ;Put scan code into AL
        MOV     AH,1                    ;Signal extended code
        JMP     DONE_READING
READ_KEY        ENDP


        END
```

Reassemble Kbd_io, by using ML /C KBD_IO.ASM. Link the four files: Test, Kbd_io, Video_io, and Cursor to try this version of READ_BYTE.

At this point, you have two problems with READ_BYTE. You cannot read the special function keys with DOS function 0Ah. Try pressing a function key when you run Test. DOS doesn't return two bytes, with the first set to zero as you might expect. Instead, the test program reports 255 for the special key (1 in AH), which means READ_BYTE didn't read any characters.

You cannot read extended codes with DOS' buffered input, using function 0Ah. This function was used so you could use the Backspace key to delete characters before the Enter key was pressed. Because you cannot read special function keys, you have to write your own READ_STRING procedure. You will have to replace function 0Ah to ensure that you can press a special function key without pressing Enter.

The other problem with DOS' function 0Ah for keyboard input has to do with the line-feed character. Press Control-Enter (line feed) after you type one character, and then try the Backspace key. You will find that you're on the next line, with no way to return to the one above. The new version of Kbd_io in the next chapter will treat the line-feed character (Control-Enter) as an ordinary character; then, pressing line feed won't move the cursor to the next line.

Before moving on to fix the problems with READ_BYTE and READ_STRING, you will write a procedure to read an unsigned decimal number. The procedure will not be used in this book, but the version of Dskpatch on the companion disk does use it so that you can, for example, ask Dskpatch to display sector number 567.

Decimal Input

If you recall, the largest unsigned decimal number that can be put into a single word is 65536. When you use READ_STRING to read a string of decimal digits, you will tell DOS to read no more than six characters (five digits and a carriage return at the end). Of course, that means READ_DECIMAL will still be able to read numbers from 65536 to 99999, even though these numbers don't fit into one word. You will have to keep watch for such numbers and return an error code if READ_DECIMAL tries to read a number larger than 65535, or if it tries to read a character that is not between zero and nine.

To convert our string of up to five digits into a word, you will use multiplication as you did in Chapter 1: You will take the first (leftmost) digit, multiply it by ten, tack on the second digit, multiply it by ten, and so on. Using this method, we could, for example, write 49856 as:

$4*10^4 + 9*10^3 + 8*10^2 + 5*10^1 + 6*10^0$

or, as you will do the calculation:

$10*(10*(10*(10*4+9) +8) +5) +6$

Of course, you must watch for errors as you do these multiplications and return with the carry flag set whenever an error occurs. How do you know when you try to read a number larger than 65535? With larger numbers, the last

MUL will overflow into the DX register. The CF flag is set when DX is not zero after a word MUL, so you can use a JC (*Jump if Carry set*) instruction to handle an error. Here is READ_DECIMAL, which also checks each digit for an error (a digit that is not between 0 and 9). Place this procedure in the file KBD_IO.ASM:

Listing 23-3 Add this procedure to KBD_IO.ASM (Complete listing in KBO_IO23.ASM)

```
        PUBLIC  READ_DECIMAL
;-----------------------------------------------------------------------;
; This procedure takes the output buffer of READ_STRING and converts    ;
; the string of decimal digits to a word.                               ;
;                                                                       ;
; Returns:      AX      Word converted from decimal                     ;
;               CF      Set if error, clear if no error                 ;
;                                                                       ;
; Uses:         READ_STRING                                             ;
; Reads:        KEYBOARD_INPUT, etc.                                    ;
; Writes:       KEYBOARD_INPUT, etc.                                    ;
;-----------------------------------------------------------------------;
READ_DECIMAL    PROC
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,6        ;Max number is 5 digits (65535)
        LEA     DX,KEYBOARD_INPUT
        CALL    READ_STRING
        MOV     CL,NUM_CHARS_READ       ;Get number of characters read
        XOR     CH,CH                   ;Set upper byte of count to 0
        CMP     CL,0                    ;Return error if no characters read
        JLE     BAD_DECIMAL_DIGIT       ;No chars read, signal error
        XOR     AX,AX                   ;Start with number set to 0
        XOR     BX,BX                   ;Start at beginning of string
CONVERT_DIGIT:
        MOV     DX,10                   ;Multiply number by 10
        MUL     DX                      ;Multiply AX by 10
        JC      BAD_DECIMAL_DIGIT       ;CF set if MUL overflowed one word
        MOV     DL,CHARS[BX]            ;Get the next digit
        SUB     DL,'0'                  ;And convert to a nibble (4 bits)
        JS      BAD_DECIMAL_DIGIT          ;Bad digit if < 0
        CMP     DL,9                       ;Is this a bad digit?
        JA      BAD_DECIMAL_DIGIT          ;Yes
        ADD     AX,DX                      ;No, so add it to number
        INC     BX                         ;Point to next character
        LOOP    CONVERT_DIGIT              ;Get the next digit
DONE_DECIMAL:
        POP     DX
        POP     CX
        POP     BX
        RET
BAD_DECIMAL_DIGIT:
        STC                                ;Set carry to signal error
        JMP     DONE_DECIMAL
READ_DECIMAL    ENDP
```

To make certain it works properly, you need to test this procedure with all the boundary conditions. A simple test program for READ_DECIMAL that uses much the same approach you used to test READ_BYTE is as follows:

Listing 23-4 Changes to TEST.ASM (Complete listing in TEST23.ASM)

```
.MODEL  SMALL

.STACK

.DATA
ENTER_PROMPT            DB      'Enter decimal number: ',0
NUMBER_READ_PROMPT      DB      'Number read: ',0
~~CHARACTER_PROMPT        DB      'Character code: ',0~~
~~SPECIAL_CHAR_PROMPT     DB      'Special character read: ',0~~

.CODE
        EXTRN   WRITE_HEX:PROC, WRITE_DECIMAL:PROC
        EXTRN   WRITE_STRING:PROC, SEND_CRLF:PROC
        EXTRN   READ_DECIMAL:PROC

TEST_READ_DECIMAL       PROC
```

continues

Listing 23-4 continued

```
        MOV     AX,DGROUP
        MOV     DS,AX

        LEA     DX,ENTER_PROMPT
        CALL    WRITE_STRING
        CALL    READ_DECIMAL
        JC      ERROR
        CALL    SEND_CRLF
        LEA     DX,NUMBER_READ_PROMPT
        CALL    WRITE_STRING
        MOV     DX,AX
        CALL    WRITE_DECIMAL
ERROR:  CALL    SEND_CRLF
~~        LEA     DX,SPECIAL_CHAR_PROMPT~~
~~        CALL    WRITE_STRING~~
~~        MOV     DL,AH~~
~~        XOR     DH,DH~~
~~        CALL    WRITE_DECIMAL~~
~~        CALL    SEND_CRLF~~

        MOV     AH,4Ch                      ;Return to DOS
        INT     21h
TEST_READ_DECIMAL       ENDP


        END     TEST_READ_DECIMAL
```

Once again, you need to link four files: Test (the preceding file), Kbd_io, Video_io, and Cursor. Try the boundary conditions, using both valid digits and invalid ones (such as A, which is not a valid decimal digit), and with such numbers as 0, 65535, and 65536. When you try a number that isn't valid, Test will exit without displaying a number after you press Enter. It only displays a number when you have typed in a valid number.

Summary

We will return to the two simple test procedures later when we discuss ways you can write your own programs. Then, you will learn how to use a slightly more advanced version of TEST.ASM to write a program that will convert numbers between hex and decimal.

Now, on to the next chapter, where you will write improved versions of READ_BYTE and READ_STRING.

C H A P T E R

Improved Keyboard Input

In this chapter you will concentrate on keyboard input where you will build a new version of READ_STRING. This new version will be able to read strings as well as special keys, such as cursor and function keys. At the end of this chapter, you will be able to use function and cursor keys in Dskpatch. Dskpatch will also require you to press Enter before it changes any bytes in SECTOR.

Files altered: KBD_IO.ASM

Disk file: KBO_IO24.ASM

24

Topics Covered

A New READ_STRING

User vs Programmer Friendly

Summary

We mentioned that we would present the development of Dskpatch just as we first wrote it—including bugs and clumsily designed procedures, some of which you have already seen. In this chapter, we will write a new version of READ_BYTE, and it will place a subtle bug into Dskpatch. In the next chapter, we will find a can of Raid to exorcise this small bug, but see if you can find it yourself first. (Hint: Carefully check all the boundary conditions for READ_BYTE when it's attached to Dskpatch.)

A New READ_STRING

The modular-design philosophy calls for short procedures so that no single procedure is too difficult to understand. The new version of READ_STRING will be an example of a procedure which is too long. It should be rewritten with more procedures, but we will leave this rewrite to you. Part III of this book is quickly drawing to an end, and you need to write a few more procedures before Dskpatch is a useful program. Right now, you can still edit only the first half of any sector, and you cannot write this sector back to the disk yet.

In this chapter you will give READ_STRING a new procedure, BACK_SPACE, to emulate the function of the Backspace key found in the DOS function 0Ah. When you press the Backspace key, BACK_SPACE will erase the last character typed from both the screen and the string in memory.

On screen, BACK_SPACE will erase the character by moving the cursor left one character, writing a space over it, and then moving left one character again. This sequence will perform the same backspace deletion provided by DOS.

In the buffer, BACK_SPACE will erase a character by changing the buffer pointer, DS:SI+BX, so it points to the next lower byte in memory. In other words, BACK_SPACE will simply decrement BX: (BX = BX - 1). The character will still be in the buffer, but your program won't see it. READ_STRING tells you how many characters it has read; if you try to read more than this number from the buffer, you will see the characters you erased. Otherwise, you won't. You have to be careful not to erase any characters when the buffer is empty. Remember that your string-data area appeared as follows:

```
CHAR_NUM_LIMIT  DB      0
NUM_CHARS_READ  DB      0
STRING          DB      80 DUP (0)
```

The string buffer starts at the second byte of this data area, or at an *offset* of 2 from the start. So BACK_SPACE won't erase a character if BX is set to 2 which is the the start of the string buffer, because the buffer is empty when BX equals 2. Place BACK_SPACE into KBD_IO.ASM as follows:

Listing 24-1 Procedure added to KBD_IO.ASM (Complete listing in KBO_IO24.ASM)

```
        PUBLIC  BACK_SPACE
        EXTRN   WRITE_CHAR:PROC
;-----------------------------------------------------------------------;
; This procedure deletes characters, one at a time, from the buffer and;
; the screen when the buffer is not empty. BACK_SPACE simply returns   ;
; when the buffer is empty.                                             ;
;                                                                       ;
; On entry:     DS:SI+BX        Most recent character still in buffer  ;
; Returns:      DS:SI+BX        Points to next most recent character   ;
;                                                                       ;
; Uses:         WRITE_CHAR                                              ;
;-----------------------------------------------------------------------;
BACK_SPACE      PROC                    ;Delete one character
        PUSH    AX
        PUSH    DX
        CMP     BX,2                    ;Is buffer empty?
        JE      END_BS                  ;Yes, read the next character
        DEC     BX                      ;Remove one character from buffer
        MOV     AH,2                    ;Remove character from screen
        MOV     DL,BS
        INT     21h
        MOV     DL,20h                  ;Write space there
        CALL    WRITE_CHAR
        MOV     DL,BS                   ;Back up again
        INT     21h
END_BS: POP     DX
        POP     AX
        RET
BACK_SPACE      ENDP
```

Let's move on to the new version of READ_STRING. The listing you will see is for only one procedure. READ_STRING is probably the longest procedure (maybe too long) you have written because it is complicated by so many possible conditions.

READ_STRING does so many things because a few more features were added. If you press the Escape key, READ_STRING clears the string buffer and remove all the characters from the screen. DOS also erases all the characters in the string buffer when you press Escape, but it does not erase any characters from the screen. Instead, it simply writes a backslash (\) character at the end of the line and moves to the next line. Our version of READ_STRING will be more versatile than the DOS READ_STRING function.

READ_STRING uses three special keys: the Backspace, Escape, and Enter keys. You could write the ASCII codes for each of these keys in READ_STRING whenever you need them; instead add a few definitions to the beginning of KBD_IO.ASM to make READ_STRING more readable. The definitions are as follows:

Listing 24-2 Additions to KBD_IO.ASM

```
.MODEL   SMALL

BS       EQU     8                          ;Backspace character
CR       EQU     13                         ;Carriage-return character
ESCAPE   EQU     27                         ;Escape character


.DATA
                             .
                             .
                             .
```

Here is READ_STRING. Although it is rather long, you can see from the listing that it is not very complicated. Replace the old version of READ_STRING in KBD_IO.ASM with the following new version:

Listing 24-3 The new READ_STRING in KBD_IO.ASM

```
        PUBLIC  READ_STRING
        EXTRN   WRITE_CHAR:PROC
;-----------------------------------------------------------------------;
; This procedure performs a function very similar to the DOS 0Ah        ;
; function. But this function will return a special character if a      ;
; function or keyboard key is pressed—no return for these keys. And     ;
; ESCAPE will erase the input and start over again.                     ;
;                                                                       ;
;       DS:DX   Address for keyboard buffer. The first byte must        ;
;               contain the maximum number of characters to read (plus ;
;               one for the return). And the second byte will be used  ;
;               by this procedure to return the number of characters    ;
;               actually read.                                          ;
;                       0       No characters read                      ;
;                       -1      One special character read              ;
;                       otherwise number actually read (not including  ;
;                               Enter key)                              ;
;                                                                       ;
; Uses:         BACK_SPACE, WRITE_CHAR, READ_KEY                        ;
;-----------------------------------------------------------------------;
READ_STRING     PROC    PROC
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                   ;Use SI for index register and
START_OVER:
        MOV     BX,2                    ;BX for offset to beginning of buffer
        CALL    READ_KEY                ;Read one key from the keyboard
        OR      AH,AH                   ;Is character extended ASCII?
        JNZ     EXTENDED                ;Yes, then process it.
STRING_NOT_EXTENDED:                    ;Extnd char is error unless buf empty
        CMP     AL,CR                   ;Is this a carriage return?
        JE      END_INPUT               ;Yes, we are done with input
        CMP     AL,BS                   ;Is it a backspace character?
        JNE     NOT_BS                  ;Nope
        CALL    BACK_SPACE              ;Yes, delete character
        CMP     BL,2                    ;Is buffer empty?
        JE      START_OVER              ;Yes, can now read extended ASCII again
        JMP     SHORT READ_NEXT_CHAR    ;No, continue reading normal characters
NOT_BS: CMP     AL,ESCAPE               ;Is it an ESC--purge buffer?
        JE      PURGE_BUFFER            ;Yes, then purge the buffer
        CMP     BL,[SI]                 ;Check to see if buffer is full
```

continues

Listing 24-3 continued

```
        JA      BUFFER_FULL             ;Buffer is full
        MOV     [SI+BX],AL              ;Else save char in buffer
        INC     BX                      ;Point to next free character in buffer
        PUSH    DX
        MOV     DL,AL                   ;Echo character to screen
        CALL    WRITE_CHAR
        POP     DX
READ_NEXT_CHAR:
        CALL    READ_KEY
        OR      AH,AH                   ;An extended ASCII char is not valid
                                        ; when the buffer is not empty
        JZ      STRING_NOT_EXTENDED     ;Char is valid

;-----------------------------------------;
; Signal an error condition by sending a beep  ;
; character to the display: chr$(7).          ;
;-----------------------------------------;
SIGNAL_ERROR:
        PUSH    DX
        MOV     DL,7                    ;Sound the bell by writing chr$(7)
        MOV     AH,2
        INT     21h
        POP     DX
        JMP     SHORT READ_NEXT_CHAR    ;Now read next character

;-----------------------------------------;
; Empty the string buffer and erase all the    ;
; characters displayed on the screen.         ;
;-----------------------------------------;
PURGE_BUFFER:
        PUSH    CX
        MOV     CL,[SI]                 ;Backspace over maximum number of
        XOR     CH,CH
PURGE_LOOP:                             ; characters in buffer. BACK_SPACE
        CALL    BACK_SPACE              ; will keep the cursor from moving too
        LOOP    PURGE_LOOP              ; far back
        POP     CX
        JMP     START_OVER              ;Can now read extended ASCII characters
                                        ; since the buffer is empty

;-----------------------------------------;
; The buffer was full, so can't read another   ;
; character. Send a beep to alert user of      ;
; buffer-full condition.                    ;
;-------------------------------------------;
BUFFER_FULL:
        JMP     SHORT SIGNAL_ERROR      ;If buffer full, just beep

;-------------------------------------------;
; Read the extended ASCII code and place this ;
; in the buffer as the only character, then  ;
; return -1 as the number of characters read. ;
;-------------------------------------------;
EXTENDED:                               ;Read an extended ASCII code
        MOV     [SI+2],AL               ;Place just this char in buffer
        MOV     BL,0FFh                 ;Num chars read = -1 for special
        JMP     SHORT END_STRING

;-------------------------------------------;
; Save the count of the number of characters ;
; read and return.                          ;
;-------------------------------------------;
END_INPUT:                              ;Done with input
        SUB     BL,2                    ;Count of characters read
END_STRING:
        MOV     [SI+1],BL               ;Return number of chars read
        POP     SI
        POP     BX
        POP     AX
        RET
READ_STRING     ENDP
```

Stepping through the procedure, you can see that READ_STRING first checks to see if you pressed a special function key. It allows you to do so only when the string is empty. For example, if you press the F3 key after pressing the *a* key, READ_STRING will ignore the F3 key and beep to tell you that you pressed a special key at the wrong time (we will fix this problem later in the chapter). You can, however, press Escape, then F3, because the Escape key causes READ_STRING to clear the string buffer.

If READ_STRING reads a carriage-return character, it places the number of characters it read into the second byte of the string area and returns. The new version of READ_BYTE looks at this byte to see how many characters READ_STRING actually read.

Next, READ_STRING checks to see if you typed a backspace character. If so, it CALLs BACK_SPACE to erase one character. If the string buffer becomes empty (BX becomes equal to 2—the start of the string buffer), then READ_STRING goes back to the start, where it can read a special key. Otherwise it just reads the next character.

Finally, READ_STRING checks for the ESCAPE character. BACK_SPACE erases characters only when there are characters in the buffer. You can clear the string buffer by calling the BACK_SPACE procedure CHAR_NUM_LIMIT times, because READ_STRING can never read more than CHAR_NUM_LIMIT characters. Any other character is stored in the string buffer and echoed to the screen with WRITE_CHAR, unless the buffer is full.

In the last chapter, READ_BYTE was changed in such a way that it couldn't read special function keys. You need to add only a few lines here to allow READ_BYTE to work with the new version of READ_STRING, which can read special function keys. The changes you need to make to READ_BYTE in KBD_IO.ASM are as follows:

Listing 24-4 Changes to READ_BYTE in KBD_IO.ASM

```
        PUBLIC  READ_BYTE
;-----------------------------------------------------------------------;
; This procedure reads a single ASCII character of a hex number.       ;
;                                                                       ;
; Returns:      AL      Character code (unless AH = 0)                  ;
;               AH      0 if read ASCII char or hex number              ;
;                       1 if read a special key                         ;
;                       -1 if no characters read                        ;
;                                                                       ;
; Uses:         HEX_TO_BYTE, STRING_TO_UPPER, READ_STRING               ;
; Reads:        KEYBOARD_INPUT, etc.                                    ;
; Writes:       KEYBOARD_INPUT, etc.                                    ;
;-----------------------------------------------------------------------;
READ_BYTE       PROC
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,3        ;Allow only two characters (plus Enter)
        LEA     DX,KEYBOARD_INPUT
        CALL    READ_STRING
        CMP     NUM_CHARS_READ,1        ;See how many characters
        JE      ASCII_INPUT             ;Just one, treat as ASCII character
        JB      NO_CHARACTERS           ;Only Enter key hit
        CMP     BYTE PTR NUM_CHARS_READ,0FFh    ;Special function key?
        JE      SPECIAL_KEY                ;Yes
        CALL    STRING_TO_UPPER            ;No, convert string to uppercase
        LEA     DX,CHARS                   ;Address of string to convert
        CALL    HEX_TO_BYTE                ;Convert string from hex to byte
        JC      NO_CHARACTERS              ;Error, so return 'no characters read'
        XOR     AH,AH                      ;Signal read one byte
DONE_READ:
        POP     DX
        RET
NO_CHARACTERS:
        XOR     AH,AH                      ;Set to 'no characters read'
        NOT     AH                         ;Return -1 in AH
        JMP     DONE_READ
ASCII_INPUT:
        MOV     AL,CHARS                   ;Load character read
        XOR     AH,AH                      ;Signal read one character
        JMP     DONE_READ
SPECIAL_KEY:
        MOV     AL,CHARS[0]                ;Return the scan code
        MOV     AH,1                       ;Signal special key with 1
        JMP     DONE_READ
READ_BYTE       ENDP
```

Dskpatch, with the new versions of READ_BYTE and READ_STRING, should be much nicer to use. Unfortunately, there is a bug here. Try to find it by running Dskpatch and by trying all the boundary conditions for READ_BYTE and HEX_TO_BYTE. (Remember that there are nine files that must be linked and converted to an EXE program: Dskpatch, Dispatch, Disp_sec, Disk_io, Video_io, Kbd_io, Phantom, Cursor, and Editor.)

User *vs* Programmer Friendly

We made a design decision in READ_STRING that made Dskpatch easier to write, but it is not friendlier to the user. Run Dskpatch and try the following: type a letter, such as *f*, then press one of the cursor keys. Dskpatch will beep at you because the READ_STRING procedure does not return control once you have started entering a hex number until you press either the Escape or the

Enter key. Unfortunately, the user probably won't know why Dskpatch is beeping at them and that creates a problem. Users also tend to become rather irritated when programs beep at them for no apparent reason.

Programs like this are *Programmer Friendly* since they are simple for the programmer to write. *User Friendly* programs, on the other hand, often require a considerable effort in programming to make them feel simple and natural. A few words of advice on writing user-friendly programs are as follows:

- Avoid beeps except to alert the user of a critical error condition (such as a disk error). There is rarely cause to beep when you press a key that isn't allowed.
- Try to keep in mind what users will want, rather than what is simple to write. Sometimes they will be one and the same, but more often than not, you will find that you have to expend additional effort and development time to write user-friendly programs.
- Try to write modeless programs. By doing so you will eliminate many error conditions such as the one we placed (artificially) into READ_STRING.
- Try out your ideas on real users, not just on other programmers who can easily figure out how your program really works. Users don't want to understand your assumptions; they want your programs to be "obvious." If a user has trouble running your program, try to understand why so you can make it easier to use.

These words of advice just scratch the surface on the issue of writing user-friendly programs. There are a number of books devoted entirely to design; we have recommended a few books in the bibliography that you will find in the last chapter of this book.

The real problem with READ_STRING is that it is modal. As soon as you type one character, you cannot do anything else until you finish typing or press Esc. What READ_STRING should do is only slightly different from what it is doing now. As soon as READ_STRING sees an extended character, it should clear the buffer and return the extended character. That way you won't need to finish entering a string before you can press an extended key. In other words, you can type a letter and then press any of the cursor keys and Dskpatch will move the cursor. A new version of READ_STRING that removes the modal behavior from the previous version is as follows:

Listing 24-5 Changes to READ_STRING in KBD_IO.ASM

```
READ_STRING     PROC     PROC
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                           ;Use SI for index register and

START_OVER:
        MOV     BX,2                            ;BX for offset to beginning of buffer

READ_LOOP:
        CALL    READ_KEY                        ;Read one key from the keyboard
        OR      AH,AH                           ;Is character extended ASCII?
        JNZ     EXTENDED                        ;Yes, then process it.
STRING_NOT_EXTENDED:                            ;No, see what char it is
        CMP     AL,CR                           ;Is this a carriage return?
        JE      END_INPUT                       ;Yes, we are done with input
        CMP     AL,BS                           ;Is it a backspace character?
        JNE     NOT_BS                          ;Nope
        CALL    BACK_SPACE                      ;Yes, delete character
        JMP     READ_LOOP                       ;Read the next character
~~        JMP     SHORT READ_NEXT_CHAR            ;No, continue reading normal characters~~

NOT_BS: CMP     AL,ESCAPE                       ;Is it an ESC--purge buffer?
~~        JE      PURGE_BUFFER                    ;Yes, then purge the buffer~~
        JNE     NOT_ESC                         ;No, put character into buffer
        CALL    PURGE_BUFFER                    ;Yes, remove all characters from buffer
        JMP     READ_LOOP                       ;Start reading characters again

NOT_ESC:
        CMP     BL,[SI]                         ;Check to see if buffer is full
        JA      BUFFER_FULL                     ;Buffer is full
        MOV     [SI+BX],AL                      ;Else save char in buffer
        INC     BX                              ;Point to next free character in buffer
        PUSH    DX
        MOV     DL,AL                           ;Echo character to screen
        CALL    WRITE_CHAR
        POP     DX
        JMP     READ_LOOP                       ;Read the next character
```

continues

Listing 24-5 continued

```
~~READ_NEXT_CHAR:~~
        ~~CALL    READ_KEY~~
        ~~OR      AH,AH                   ;An extended ASCII char is not valid~~
                                          ~~; when the buffer is not empty~~
        ~~JZ      STRING_NOT_EXTENDED     ;Char is valid~~

;-----------------------------------------------;
; Signal an error condition by sending a beep   ;
; character to the display: chr$(7).            ;
;-----------------------------------------------;
SIGNAL_ERROR:
        PUSH    DX
        MOV     DL,7                    ;Sound the bell by writing chr$(7)
        MOV     AH,2
        INT     21h
        POP     DX
        JMP     SHORT READ_LOOP         ;Now read next character

~~;-----------------------------------------------;~~
~~; Empty the string buffer and erase all the     ;~~
~~; characters displayed on the screen.           ;~~
~~;-----------------------------------------------;~~
~~PURGE_BUFFER:~~
        ~~PUSH    CX~~
        ~~MOV     CL,[SI]                 ;Backspace over maximum number of~~
        ~~XOR     CH,CH~~
~~PURGE_LOOP:                             ; characters in buffer. BACK_SPACE~~
        ~~CALL    BACK_SPACE              ; will keep the cursor from moving too~~
        ~~LOOP    PURGE_LOOP              ; far back~~
        ~~POP     CX~~
        ~~JMP     START_OVER              ;Can now read extended ASCII characters~~
                                          ~~; since the buffer is empty~~
;-----------------------------------------------;
; The buffer was full, so can't read another    ;
; character. Send a beep to alert user of       ;
; buffer-full condition.                        ;
;-----------------------------------------------;
BUFFER_FULL:
        JMP     SHORT SIGNAL_ERROR      ;If buffer full, just beep
```

```
;-----------------------------------------;
; Read the extended ASCII code and place this  ;
; in the buffer as the only character, then    ;
; return -1 as the number of characters read.  ;
;-----------------------------------------;
EXTENDED:                               ;Read an extended ASCII code
        CALL    PURGE_BUFFER            ;Remove any chars from buffer
        MOV     [SI+2],AL               ;Place just this char in buffer
        MOV     BL,0FFh                 ;Num chars read = -1 for special
        JMP     SHORT END_STRING

;-----------------------------------------;
; Save the count of the number of characters   ;
; read and return.                             ;
;-----------------------------------------;
END_INPUT:                              ;Done with input
        SUB     BL,2                    ;Count of characters read
END_STRING:
        MOV     [SI+1],BL               ;Return number of chars read
        POP     SI
        POP     BX
        POP     AX
        RET
READ_STRING     ENDP


;----------------------------------------------------------------------;
; This subroutine is used by READ_STRING to clear the contents of the  ;
; input buffer.                                                        ;
;                                                                      ;
;       DS:SI   Points to the input buffer for READ_STRING             ;
;----------------------------------------------------------------------;
PURGE_BUFFER    PROC
        PUSH    CX
        MOV     CL,[SI]                 ;Backspace over maximum number of
        XOR     CH,CH
PURGE_LOOP:                             ; characters in buffer. BACK_SPACE
        CALL    BACK_SPACE              ; will keep the cursor from moving too
        LOOP    PURGE_LOOP              ; far back
        POP     CX
        RET
PURGE_BUFFER    ENDP
```

You will notice that more code was removed than added. This is a good sign because solving a problem by simplifying the code results in more reliable as well as usable programs.

Summary

You wrote a new version of READ_STRING in this chapter that allowed you to read special characters again, in addition to strings. With the exception of the small bug that you will find and fix in the next chapter, READ_STRING works as advertised. This discussion covered several problems with READ_STRING, one being that it is too long and complicated and should be rewritten to be more modular.

Finally, you learned that READ_STRING was not user friendly since it beeped when you tried to move the cursor after you have started to type a hex number. You fixed both of these problems at the end of this chapter. Now it is time to remove the bug that lurks in Dskpatch.

C H A P T E R

In Search of Bugs

In this chapter you will learn how to fix a small bug that appeared in Dskpatch when you put all the pieces together. See if you can find the bug by trying all the boundary conditions for the hex input prompt.

Files altered: DISPATCH.ASM

Disk file: DISPAT25.ASM

25

Topics Covered

If you try the new version of Dskpatch with *ag*, which is not a valid hex number, you will notice that Dskpatch does not do anything when you press the Enter key. Since the string *ag* is not a hex number, there is nothing wrong with Dskpatch ignoring it, but the program should, at least, erase it from the screen.

This error is the sort we can find only by thoroughly checking the boundary conditions of a program; not just the pieces, but the entire program. The bug here isn't the fault of READ_BYTE, even though it appeared when you rewrote that procedure. Rather, the problem is in the way we wrote DISPATCHER and EDIT_BYTE.

EDIT_BYTE is designed so it calls WRITE_PROMPT_LINE to rewrite the editor prompt line and clear the rest of the line. This will remove any character you typed. If you type a string like *ag*, READ_BYTE reports that it read a string of zero length, and DISPATCH does not call EDIT_BYTE. What is the solution?

Fixing DISPATCHER

There are actually two ways to solve this problem. The best solution would be to rewrite Dskpatch to be more modular, and to redesign DISPATCHER. We won't do that. Remember: Programs are never complete, but you have to stop somewhere. Instead, you will add a fix to DISPATCHER so it will rewrite the prompt line whenever READ_BYTE reads a string of zero length. Modifications to DISPATCHER (in DISPATCH.ASM) to fix the bug are as follows:

Listing 25-1 Changes to DISPATCHER in DISPATCH.ASM (Complete listing in DISPAT25.ASM)

```
        PUBLIC  DISPATCHER
        EXTRN   READ_BYTE:PROC, EDIT_BYTE:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC
.DATA
        EXTRN   EDITOR_PROMPT:BYTE
.CODE
```

```
;----------------------------------------------------------------------;
; This is the central dispatcher.  During normal editing and viewing,   ;
; this procedure reads characters from the keyboard and, if the character;
; is a command key (such as a cursor key), DISPATCHER calls the          ;
; procedures that do the actual work.  This dispatching is done for      ;
; special keys listed in the table DISPATCH_TABLE, where the procedure   ;
; addresses are stored just after the key names.                         ;
;   If the character is not a special key, then it should be placed      ;
; directly into the sector buffer--this is the editing mode.             ;
;                                                                        ;
; Uses:         READ_BYTE, EDIT_BYTE, WRITE_PROMPT_LINE                  ;
; Reads:        EDITOR_PROMPT                                            ;
;----------------------------------------------------------------------;
DISPATCHER      PROC
        PUSH    AX
        PUSH    BX
        PUSH    DX
DISPATCH_LOOP:
        CALL    READ_BYTE                 ;Read character into AX
        OR      AH,AH                     ;AX = -1 if no character read, 1
                                          ; for an extended code.
        JS      NO_CHARS_READ             ;No character read, try again
        JNZ     SPECIAL_KEY               ;Read extended code
        MOV     DL,AL
        CALL    EDIT_BYTE                 ;Was normal character, edit byte
        JMP     DISPATCH_LOOP             ;Read another character

SPECIAL_KEY:
        CMP     AL,68                     ;F10--exit?
        JE      END_DISPATCH              ;Yes, leave
                                          ;Use BX to look through table
        LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
        CMP     BYTE PTR [BX],0           ;End of table?
        JE      NOT_IN_TABLE              ;Yes, key was not in the table
        CMP     AL,[BX]                   ;Is it this table entry?
        JE      DISPATCH                  ;Yes, then dispatch
        ADD     BX,3                      ;No, try next entry
        JMP     SPECIAL_LOOP              ;Check next table entry

DISPATCH:
        INC     BX                        ;Point to address of procedure
```

continues

Listing 25-1 continued

```
         CALL     WORD PTR [BX]          ;Call procedure
         JMP      DISPATCH_LOOP          ;Wait for another key

NOT_IN_TABLE:                            ;Do nothing, just read next character
         JMP      DISPATCH_LOOP

NO_CHARS_READ:
         LEA      DX,EDITOR_PROMPT
         CALL     WRITE_PROMPT_LINE      ;Erase any invalid characters typed
         JMP      DISPATCH_LOOP          ;Try again

END_DISPATCH:
         POP      DX
         POP      BX
         POP      AX
         RET
DISPATCHER        ENDP
```

This bug fix does not create any great problems, but it does make DISPATCHER slightly less elegant. Elegance is a virtue to strive for. Elegance and clarity often go hand in hand, and the rules of modular design are aimed at increasing elegance.

Summary

DISPATCHER is elegant because it is such a simple solution to a problem. Rather than using many comparisons for each special character you might type, you built a table that can be searched. Doing so made DISPATCHER simpler and more reliable than a program containing different instructions for each possible condition that might arise. By adding the small fix, we complicated DISPATCHER; not by much in this case, but some bugs might require you to really complicate a procedure.

If you find yourself adding fixes that make a procedure too complicated, rewrite whichever procedures you must to remove this complexity. Always check the boundary conditions before and after you add a procedure to your main program. You will save yourself a lot of debugging effort if you do.

We cannot overemphasize the importance of testing procedures with boundary conditions and of following the rules of modular design. Both techniques lead to better and more reliable programs. The next chapter will cover another method for debugging programs.

C H A P T E R

Writing Modified Sectors

In this chapter you will learn how to build and navigate a road map for your programs. You will also learn how to use the CodeView and Turbo Debugger source-level debuggers to trace through your programs. You will also add a new function to Dskpatch so it can write sectors to the disk.

Files altered: DISPATCH.ASM, DISK_IO.ASM

Disk files: DISPAT26.ASM, DISK_I26.ASM, MAKEFILE, LINKINFO

26

Topics Covered

In this chapter, you will build a procedure to write a modified sector back to disk. In the next chapter, you will write a procedure to show the second half of a sector.

Writing to the Disk

Writing a modified sector back to the disk can be disastrous if it is not done intentionally. All of Dskpatch's functions have depended on the function keys F3, F4, and F10, and on the cursor keys. But any of these keys could be pressed quite by accident. Fortunately, you can use the shifted keys without this happening. We have chosen the shifted F2 key for writing a disk sector because F2 is often used in programs to save changes. This will prevent you from writing a sector back to disk unless you really want to.

The following changes should be made to DISPATCH.ASM to add WRITE_SECTOR to the table.

Listing 26-1 Changes to DISPATCH.ASM (Complete listing in DISPAT26.ASM)

```
.CODE
        EXTRN   NEXT_SECTOR:PROC                        ;In DISK_IO.ASM
        EXTRN   PREVIOUS_SECTOR:PROC                    ;In DISK_IO.ASM
        EXTRN   PHANTOM_UP:PROC, PHANTOM_DOWN:PROC      ;In PHANTOM.ASM
        EXTRN   PHANTOM_LEFT:PROC, PHANTOM_RIGHT:PROC
        EXTRN   WRITE_SECTOR:PROC                       ;In DISK_IO.ASM
.DATA
;----------------------------------------------------------------------;
; This table contains the legal extended ASCII keys and the addresses  ;
; of the procedures that should be called when each key is pressed.    ;
;                                                                      ;
; The format of the table is                                           ;
;               DB      72              ;Extended code for cursor up   ;
;               DW      OFFSET PHANTOM_UP                              ;
;----------------------------------------------------------------------;
DISPATCH_TABLE  LABEL   BYTE
        DB      61                              ;F3
        DW      OFFSET _TEXT:PREVIOUS_SECTOR
        DB      62                              ;F4
        DW      OFFSET _TEXT:NEXT_SECTOR
        DB      72                              ;Cursor up
        DW      OFFSET _TEXT:PHANTOM_UP
        DB      80                              ;Cursor down
        DW      OFFSET _TEXT:PHANTOM_DOWN
        DB      75                              ;Cursor left
        DW      OFFSET _TEXT:PHANTOM_LEFT
        DB      77                              ;Cursor right
        DW      OFFSET _TEXT:PHANTOM_RIGHT
        DB      85                              ;Shift F2
        DW      OFFSET _TEXT:WRITE_SECTOR
        DB      0                               ;End of the table
```

WRITE_SECTOR itself is almost identical to READ_SECTOR. The only change is that you wish to write, rather than read, a sector. Whereas the INT 25h asks DOS to read one sector, its companion function, INT 26h, asks DOS to write a sector to the disk. Place WRITE_Sector into DISK_IO.ASM as follows:

Listing 26-2 Procedure added to DISK_IO.ASM (Complete listing in DISK_I26.ASM)

```
        PUBLIC  WRITE_SECTOR
;-----------------------------------------------------------------------;
; This procedure writes the sector back to the disk.                    ;
;                                                                       ;
; Reads:        DISK_DRIVE_NO, CURRENT_SECTOR_NO, SECTOR                ;
;-----------------------------------------------------------------------;
WRITE_SECTOR    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AL,DISK_DRIVE_NO        ;Drive number
        MOV     CX,1                    ;Write 1 sector
        MOV     DX,CURRENT_SECTOR_NO    ;Logical sector
        LEA     BX,SECTOR
        INT     26h                     ;Write the sector to disk
        POPF                            ;Discard the flag information
        POP     DX
```

continues

Listing 26-2 continued

```
        POP     CX
        POP     BX
        POP     AX
        RET
WRITE_SECTOR    ENDP
```

Now reassemble both Dispatch and Disk_io, but don't try Dskpatch's write function just yet. Find an old disk you don't need and put it in drive A. Run Dskpatch, which will read the first sector from your scratch disk in drive A. Before you go on, make sure this is a scratch disk you have no qualms about destroying.

Change one byte in your sector display. Make a note of the one you changed and what value it had before. Then press the shifted F2 key. You will see the red drive light come on; you have just written a modified sector back to drive A.

Next, press F4 to read the next sector (sector 1), then F3 to read the previous sector (your original sector, number 0). You should see the modified sector back again. Restore the number you changed in this sector and write it back to Drive A to restore the integrity of your scratch disk.

More Debugging Techniques

What would happen if you had made a small error in the program? Dskpatch is large enough that you might have problems using Debug to find the bug. In addition, Dskpatch is composed of nine different files that must be linked to form DSKPATCH.EXE. How do you find one procedure in this large program found without tracing slowly through much of the program? As you will learn in this chapter, there are two ways to find procedures: by using a road map from LINK, or by using a source-level debugger, such as Microsoft's CodeView or Borland's Turbo Debugger.

When we (the authors) originally wrote Dskpatch, something went wrong when we added WRITE_SECTOR; pressing the Shift-F2 key caused our machine to hang. But we couldn't find anything wrong with WRITE_SECTOR and the only other changes were to DISPATCH_TABLE. Everything appeared to be correct. Finally, we traced the bug to a faulty definition in the dispatcher. The bug turned out to be an error in the DISPATCH_TABLE entry for WRITE_SECTOR. Somehow, we had typed a DW rather than a DB in the table, so WRITE_SECTOR's address was stored one byte higher in memory than it should have been. You can see the bug shown against a gray background as follows:

```
DISPATCH_TABLE  LABEL   BYTE
                  .
                  .
                  .
        DB      77                              ;Cursor right
        DW      OFFSET _TEXT:PHANTOM_RIGHT
        DW      85                              ;Shift F2
        DW      OFFSET _TEXT:WRITE_SECTOR
        DB      0                               ;End of the table
DATA_SEG        ENDS
```

As an exercise in debugging, make this change to your file DISPATCH.ASM (diskfile DISPAT26.ASM), then follow the directions in the next section.

Building a Road Map

Now you will learn how to use LINK to build a map of Dskpatch. This map will help you find procedures and variables in memory. The LINK command you have used in the make file has grown to be fairly long, as follows, and you will add more to it.

```
LINK DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR DISPATCH KBD_IO PHANTOM EDITOR;
```

You will not have to keep typing file after file because LINK allows you to supply an automatic response file containing all the information. We will call the file LINKINFO and type the following:

```
LINK @LINKINFO
```

With the file names used so far, LINKINFO appears as follows:

```
DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR +
DISPATCH KBD_IO PHANTOM EDITOR
```

The plus (+) at the end of the first line tells LINK to continue reading file names from the next line.

You can add more information that tells LINK to create a map of the procedures and variables in the program to this simple Linkinfo file. The entire LINKINFO file is as follows:

```
DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR +
DISPATCH KBD_IO PHANTOM EDITOR
DSKPATCH
DSKPATCH /MAP;
```

The last two lines are new parameters. The first, DSKPATCH, tells LINK you want the .EXE file to be named DSKPATCH.EXE; the second new line tells LINK to create a listing file called DSKPATCH.MAP to create the road map. The /map switch tells LINK to provide a list of all the procedures and variables that you have declared to be public. (Incidentally, the Programmer's Workbench provides these same abilities with a nice user interface. This was covered in Chapter 13.)

Create the map file by relinking Dskpatch with this LINKINFO response file. You will probably want to change your Makefile so it uses LINK @LINKINFO instead of the long link line. The map file produced by the linker is about 140 lines long. The file is too long to be reproduced in its entirety, so we reproduced only the parts that are of particular interest. A partial listing of the map file, DSKPATCH.MAP, is as follows:

```
 Start  Stop   Length Name                   Class
 00000H 005B6H 005B7H _TEXT                  CODE
 005B8H 006A9H 000F2H _DATA                  DATA
 006AAH 026A9H 02000H _BSS                   BSS
 026B0H 02AAFH 00400H STACK                  STACK

 Origin   Group
 005B:0   DGROUP

  Address         Publics by Name
```

```
0000:03DB         BACK_SPACE
0000:0274         CLEAR_SCREEN
0000:02B5         CLEAR_TO_END_OF_LINE
0000:033D         CONVERT_HEX_DIGIT
005B:000A         CURRENT_SECTOR_NO
0000:0296         CURSOR_RIGHT
005B:000C         DISK_DRIVE_NO
0000:0010         DISK_PATCH
0000:02E2         DISPATCHER
0000:0127         DISP_HALF_SECTOR
                  .
                  .
                  .
0000:01E1         WRITE_HEX_DIGIT
0000:0255         WRITE_PATTERN
0000:0534         WRITE_PHANTOM
0000:019F         WRITE_PROMPT_LINE
0000:0081         WRITE_SECTOR
0000:01B2         WRITE_STRING
0000:00F0         WRITE_TOP_HEX_NUMBERS
0000:0572         WRITE_TO_MEMORY
005B:00FA         _edata
005B:2100         _end

 Address           Publics by Value

0000:0010         DISK_PATCH
0000:002E         PREVIOUS_SECTOR
0000:004D         NEXT_SECTOR
0000:0068         READ_SECTOR
0000:0081         WRITE_SECTOR
0000:009A         INIT_SEC_DISP
0000:00C4         WRITE_HEADER
0000:00F0         WRITE_TOP_HEX_NUMBERS
0000:0127         DISP_HALF_SECTOR
                  .
                  .
                  .
005B:000D         LINES_BEFORE_SECTOR
005B:000E         HEADER_LINE_NO
005B:000F         HEADER_PART_1
005B:0015         HEADER_PART_2
005B:0026         PROMPT_LINE_NO
005B:0027         EDITOR_PROMPT
005B:00F8         PHANTOM_CURSOR_X
```

```
 005B:00F9       PHANTOM_CURSOR_Y
 005B:00FA       SECTOR
 005B:00FA       _edata
 005B:2100       _end
Program entry point at 0000:0010
```

There are three main parts to this load map (so called because it tells you where your procedures are loaded in memory). The first shows a list of segments in the program. Dskpatch has several segments: _TEXT (which contains all your code) and _DATA, _BSS, and STACK, which are grouped together into the group DGROUP, and contain all your data. For those of you interested in more detail, _DATA contains all the memory variables defined in the .DATA segment (such as HEADER_LINE_NO), _BSS contains variables defined in the .DATA? segment (such as SECTOR), and STACK contains the stack defined by .STACK. You may see slightly different numbers in the load map if your procedures are in a different order than our procedures (you can check the order in Appendix B).

The next part of the load map shows the public procedures and variables listed in alphabetic order. LINK lists only those procedures and variables you have declared to be PUBLIC—visible to the outside world. If you are debugging a long program, you may want to declare all procedures and variables to be public, so you can find them in this map.

The final section of the map lists all the procedures and memory variables again, but this time in the order they appear in memory. Both of these lists include the memory address for each PUBLIC procedure or variable. If you check this list, you will find that the procedure DISPATCHER starts at address 2E2h. You will use this address to track down the bug in Dskpatch.

Tracking Down Bugs

If you were to try running the version of Dskpatch with the bug in it, you would find that everything works with the exception of Shift-F2 which caused Dskpatch to hang on our machine. You don't want to try Shift-F2; there is no telling what it will do on your machine.

Since everything worked (and works now) except for Shift-F2, our first guess when we wrote the program was that we had introduced a bug into

WRITE_SECTOR. To find this bug, we could start debugging Dskpatch by tracing through WRITE_SECTOR. Instead, we will take a somewhat different tack.

You know that DISPATCHER works correctly, because everything else (the cursor keys, F3, F4, and F10) all works correctly. That means DISPATCHER is a good starting point to search for the bug in Dskpatch. In other words, start your bug search with code you know works properly. If you look at the program listing for DISPATCHER (in Chapter 25), you will see that the following instruction is the heart of DISPATCHER, because it calls all the other routines.

```
CALL    WORD PTR [BX]
```

In particular, this CALL instruction will call WRITE_SECTOR when you press Shift-F2. Let's start the search here.

Warning

For various reasons, the DISPATCHER procedure in your program may be at an address other than 2E2h. Instead of using the addresses below exactly as you see them, you will need to choose the correct address. For DISPATCHER, look in the Dskpatch.map file to find its address. Then in the following code, use the addresses you see in Debug, rather than the addresses shown in this book.

You will use Debug to start Dskpatch with a breakpoint set on this instruction. That means you need the address of this instruction. It can be found by unassembling DISPATCHER which starts at 2E2h. After a U 2E2, followed by another U command, you should see the CALL command as follows:

```
                        .
                        .
                        .
3AC1:0308 EBF2          JMP     02FC
3AC1:030A 43            INC     BX
3AC1:030B FF17          CALL    [BX]
3AC1:030D EBD6          JMP     02E5
                        .
                        .
                        .
```

Now that you know the CALL instruction is at location 30Bh, you can set a breakpoint at this address, then single-step into and through WRITE_SECTOR.

First, use the command G 30B to execute Dskpatch up to this instruction. You will see Dskpatch start up and then wait for you to type a command. Press Shift-F2, since this is the command that is causing problems. You will see the following:

```
-G 30B



AX=0155  BX=00A1  CX=06AC  DX=0027  SP=03F8  BP=0000  SI=0000  DI=0000
DS=3B1C  ES=3AB1  SS=3D2C  CS=3AC1  IP=030B   NV UP EI PL NZ NA PO NC
3AC1:030B FF17          CALL    [BX] DS:00A1=8100
-
```

At this point the BX register is pointing to a word that should contain the address of WRITE_SECTOR. Let's see if it does.

```
-D A1 L 2
3B1C:00A0     00 81                                                ..
-
```

In other words, you are trying to CALL a procedure located at 8100h (remember the lower byte is displayed first). But if you look at the memory map, you can see that WRITE_SECTOR should be at 81h. In fact, we can also tell from this load map that there aren't any procedures at 8100h; the address is totally wrong.

In your original bug-hunting, once we discovered that this address was wrong, it didn't take us very long to find the error. We knew that DISPATCHER and the table were basically sound because all the other keys worked, so we took a closer look at the data for Shift-F2 and found the DW where we should have had a DB. Having a road map makes debugging much simpler. Now let's look at some more powerful debugging tools.

Source-Level Debugging

Microsoft and Borland have been working hard to provide the ultimate in programming tools. Microsoft's CodeView and Borland's Turbo Debugger are both debuggers of a type called Source-Level Debuggers. In other words, whereas Debug shows you just addresses in CALLs and JMPs, these two debuggers show you the actual source code.

You may only want to read one of the next two sections; since one section covers Microsoft's CodeView and the other Borland's Turbo Debugger, there is some repetition of material between the two sections.

Microsoft's CodeView

CodeView is the older of the two debuggers, having been introduced onto the market in 1986, about two years before Borland's Turbo Debugger. It is now included with every Microsoft Macro Assembler package (we are using version 6.0) as well as most of their other language products. As you will see in this section, CodeView is so useful that you may want to consider upgrading your macro assembler if you don't already have the latest version.

CodeView shares some similarities with Debug, since Microsoft wrote both programs. But there are more differences than similarities. We will use two of the new features here: source-level debugging and screen swapping.

Source-level debugging lets you see the actual source code complete with comments, rather than just instructions and addresses in your display. For example, if you use Debug to unassemble the first line in Dskpatch, you will see the following:

```
3AC1:0010 B81C3B        MOV     AX,3B1C
```

With CodeView, on the other hand, you will see the following (as you can also see in Figure 26-1):

```
MOV     AX,DGROUP                   ;Put data segment into AX
```

The second new feature, screen swapping, is handy for debugging Dskpatch. Dskpatch moves the cursor around the screen, writing in different places. In the last section, where we used Debug, Debug started writing to this same screen and eventually the Dskpatch screen was lost.

CodeView, however, maintains two separate screens: one for Dskpatch and one for itself. Whenever Dskpatch is active, you see its screen; whenever

CodeView is active, you see *its* screen. You will get a clearer idea of screen swapping as you run through the following examples.

Before you can use CodeView's symbolic debugging features you need to tell both the assembler and the linker to save debugging information. This can be done with the /Zi switch in the assembler and the /CODEVIEW switch in the linker.

Modify each line in your MAKEFILE (or reassemble each file by hand) so it has the /Zi switch before each file name and modify MAKEFILE so it uses a response file for LINK as follows:

Listing 26-3 Changes made to MAKEFILE

```
dskpatch.exe:   dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
                dispatch.obj kbd_io.obj phantom.obj editor.obj
        link @linkinfo

dskpatch.obj:   dskpatch.asm
        ml /c /Zi dskpatch.asm

disk_io.obj:    disk_io.asm
        ml /c /Zi disk_io.asm
                .
                .
                .
```

Then change the linker response file LINKINFO as follows:

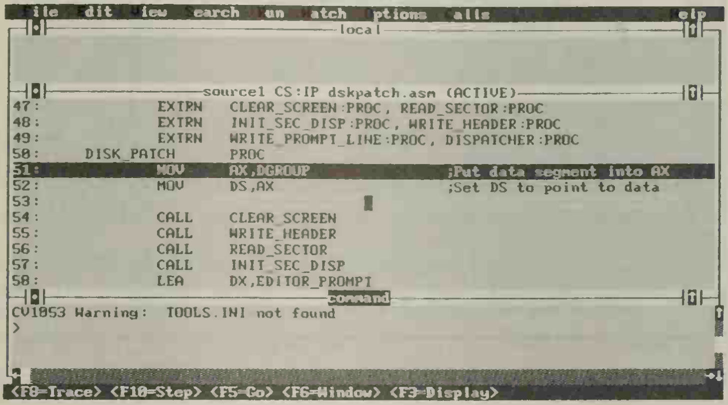
Listing 26-4 Changes to the response file LINKINFO

```
dskpatch disk_io disp_sec video_io cursor +
dispatch kbd_io phantom editor
dskpatch
dskpatch /CODEVIEW;
```

Finally, delete all the *.obj files and remake Dskpatch.exe (alternatively, you can type NMAKE /A to have NMake reassemble everything). Now you are ready to start CodeView. You should see a display like the one in Figure 26-1 when you type the following:

```
C>CV DSKPATCH
```

Notice that you are viewing the actual source file! This is why CodeView is known as a source-level debugger.


File Edit View Search Run Watch Options Calls Help
local
source1 CS:IP dskpatch.asm (ACTIVE)
command
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>

```
            EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
            EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
            EXTRN   WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
    DISK_PATCH      PROC
            MOV     AX,DGROUP                   ;Put data segment into AX
            MOV     DS,AX                       ;Set DS to point to data

            CALL    CLEAR_SCREEN
            CALL    WRITE_HEADER
            CALL    READ_SECTOR
            CALL    INIT_SEC_DISP
            LEA     DX,EDITOR_PROMPT
```

```
CV1053 Warning:  TOOLS.INI not found
>
```

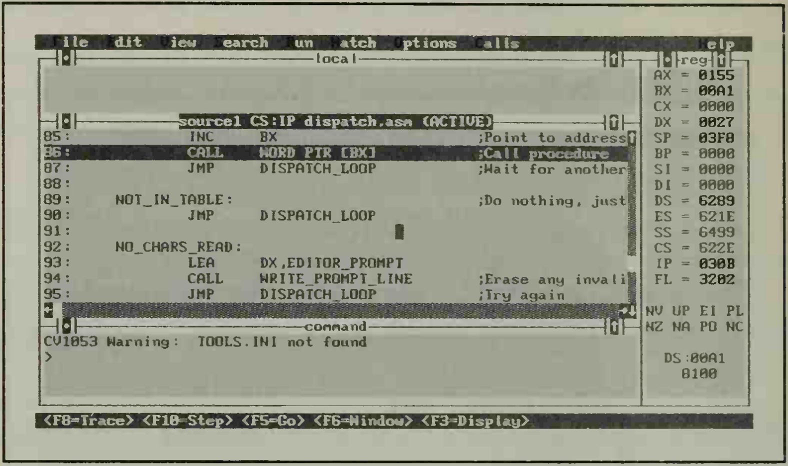
Figure 26-1: The initial view of Dskpatch.exe inside CodeView.

Now that you have CodeView up and running, you can look at the procedure DISPATCHER without knowing where it is. Press Alt-S (to pull down the Search menu), then L (Label/Function...) to search for a label. Next, type *dispatcher* into the dialog box that pops up and press Enter to see the code for DISPATCHER. Finally, use the cursor keys (or the Page Down key) to scroll to the CALL WORD PTR [BX] instruction.

Once you have the cursor on the line with the CALL WORD PTR [BX] instruction, press F7 (which will run the program until it reaches the CALL). You will see Dskpatch draw its screen. You will be returned to CodeView after you press Shift-F2. This time you won't see any of Dskpatch's screen because CodeView swapped screens. To flip back to the Dskpatch screen, press the F4 key. Once you are looking at Dskpatch's screen, pressing any key will return you to CodeView's screen.

Press the F2 key to display the register window, on the right side of CodeView's screen as in Figure 26-2, if it is not already visible. If you look on the lower right part of the register screen , you will see two short lines as follows:

```
DS:00A1
  8100
```


File Edit View Search Run Watch Options Calls Help
local
reg
source1 CS:IP dispatch.asm (ACTIVE)
command
<F8=Trace> <F10=Step> <F5=Go> <F6=Window> <F3=Display>

```
            INC     BX                          ;Point to address
            CALL    WORD PTR [BX]               ;Call procedure
            JMP     DISPATCH_LOOP               ;Wait for another
                                                
      NOT_IN_TABLE:                             ;Do nothing, just
            JMP     DISPATCH_LOOP
                                                
      NO_CHARS_READ:
            LEA     DX,EDITOR_PROMPT
            CALL    WRITE_PROMPT_LINE           ;Erase any invali
            JMP     DISPATCH_LOOP               ;Try again

CV1053 Warning:  TOOLS.INI not found
>
```

```
AX = 0155
BX = 00A1
CX = 0000
DX = 0027
SP = 03F8
BP = 0000
SI = 0000
DI = 0000
DS = 6289
ES = 621E
SS = 6499
CS = 622E
IP = 030B
FL = 3202

NV UP EI PL
NZ NA PO NC

 DS:00A1
   0100
```

Figure 26-2: CodeView after the F7 (Go) command and then F2 (Register).

This area of the display is used to show the value in memory pointed to by the memory reference in the current instruction. This is the CALL instruction under the inverse-video cursor bar. In this case, the single memory reference is to the value at memory location [BX]. As you can see, 8100 is the value you found by using Debug with the help of Link's memory map. But here you found the value much more quickly.

Type Alt-F (to pull down the File menu) and X (eXit) to exit from CodeView. You may want to skip the next section and go directly to the Summary.

Warning

Don't forget to change the DW back to a DB in Dispatch.asm.

You may also want to change back the linkinfo file. You added the /CODEVIEW switch so Link would add the debugging information to the EXE file. But this debugging information makes the .EXE file a bit larger (Dskpatch is about 11K with debugging information, and about 2K without). In any case, you will probably want to remove the /CODEVIEW switch before you give your programs to other people.

Borland's Turbo Debugger

Turbo Debugger shares few similarities with Debug. As you will see in this section, Turbo Debugger uses Borland's multiple-window style of user interface as opposed to Debug's command-line interface. Borland has also added many debugging features that are not present in Debug. You will use two of the new features here: source-level debugging and screen swapping.

Source-level debugging lets you see the actual source code complete with comments, rather than just instructions and addresses, in the display. For example, if you use Debug to unassemble the first line in Dskpatch, you will see the following:

```
3AC1:0010 B81C3B        MOV     AX,3B1C
```

With Turbo Debugger, you will see the following (as you can also see in Figure 26-3):

```
MOV     AX,DGROUP                  ;Put data segment into AX
```

The second new feature, screen swapping, is handy for debugging Dskpatch. Dskpatch moves the cursor around the screen, writing in different places. In the last section where you used Debug, it started writing to this same screen and you eventually lost the Dskpatch screen.

Turbo Debugger, however, maintains two separate screens: one for Dskpatch and one for itself. Whenever Dskpatch is active, you see its screen; whenever Turbo Debugger is active, you see *its* screen.

Before you can use Turbo Debugger's symbolic debugging features you need to tell both the assembler and the linker to save debugging information. You do this with the /zi switch in the assembler and the /v switch in the linker.

Modify each line in your Makefile (or reassemble each file by hand) so it has the /zi switch before the file name. And modify Makefile so it uses a response file for TLINK (notice that we are using TLINK):

Listing 26-5 Changes made to Makefile

```
dskpatch.exe:   dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
                dispatch.obj kbd_io.obj phantom.obj editor.obj
        tlink @linkinfo

dskpatch.obj:   dskpatch.asm
        tasm /zi dskpatch.asm

disk_io.obj:    disk_io.asm
        tasm /zi disk_io.asm
                .
                .
                .
```

Then change the linker response file LINKINFO as follows:

Listing 26-6 Changes to the response file LINKINFO

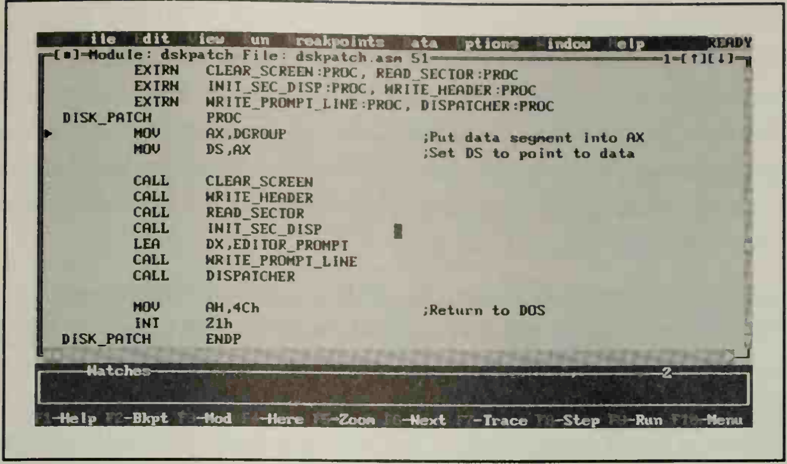
```
dskpatch disk_io disp_sec video_io cursor +
dispatch kbd_io phantom editor
dskpatch
dskpatch /v;
```

Finally, delete all the *.obj files and remake Dskpatch.exe (or type "make -B" to rebuild everything—the B must be uppercase).

Now you are ready to start Turbo Debugger. Type the following and you should see a display like the one in Figure 26-3:

```
C>TD DSKPATCH
```

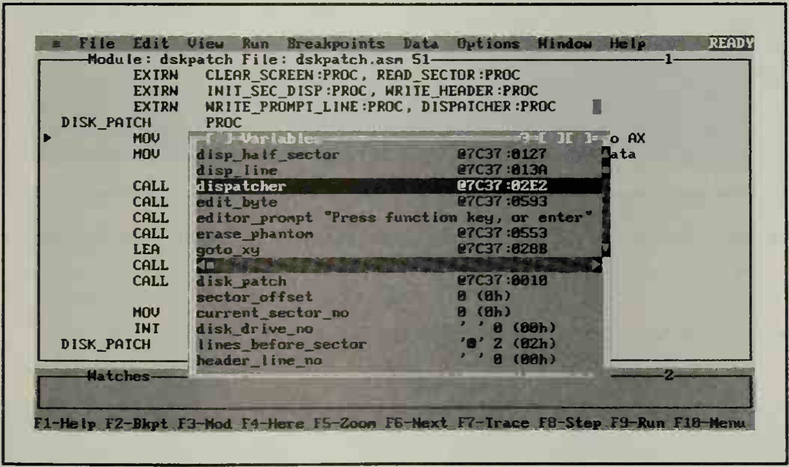
Notice that you are viewing the actual source file! This is why Turbo Debugger is known as a source-level debugger.



```
 ≡ File  Edit  View  Run  Breakpoints  Data  Options  Window  Help          READY
 [•]=Module: dskpatch File: dskpatch.asm 51=======================1=[↑][↓]=
          EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
          EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
          EXTRN   WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
  DISK_PATCH      PROC
 ►        MOV     AX,DGROUP                   ;Put data segment into AX
          MOV     DS,AX                       ;Set DS to point to data

          CALL    CLEAR_SCREEN
          CALL    WRITE_HEADER
          CALL    READ_SECTOR
          CALL    INIT_SEC_DISP
          LEA     DX,EDITOR_PROMPT
          CALL    WRITE_PROMPT_LINE
          CALL    DISPATCHER

          MOV     AH,4Ch                      ;Return to DOS
          INT     21h
  DISK_PATCH      ENDP
 ──Watches──────────────────────────────────────────────────────────2──

 F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 26-3: The initial view of Dskpatch.exe inside Turbo Debugger.

Now that Turbo Debugger is up and running, you can look at the procedure DISPATCHER without knowing where it is. Press Alt-V to pull down the View menu, followed by V to show the variable window (Figure 26-4). Use the cursor-up and -down keys to move the cursor to dispatcher; press Enter to show the code for DISPATCHER. You can then use the cursor keys to scroll to the CALL Word Ptr [BX] instruction on the second page.

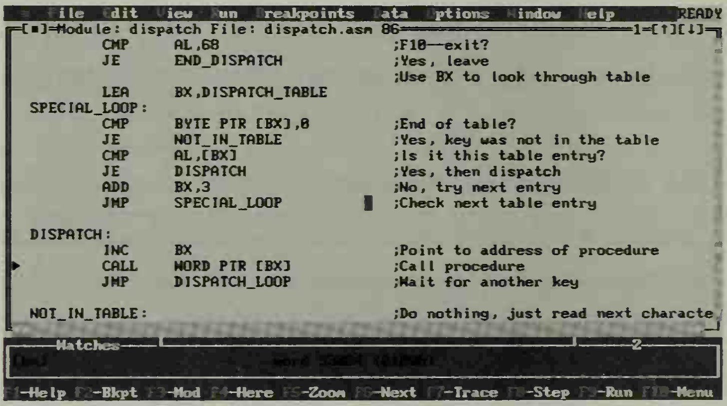


```
 ≡ File  Edit  View  Run  Breakpoints  Data  Options  Window  Help          READY
 ──Module: dskpatch File: dskpatch.asm 51──────────────────────────1──
          EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
          EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
          EXTRN   WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
  DISK_PATCH      PROC
 ►        MOV    =[ ]=Variables=====================3=[ ][ ]= o AX
          MOV     disp_half_sector          @7C37:0127        ata
                  disp_line                 @7C37:013A
          CALL    dispatcher                @7C37:02E2
          CALL    edit_byte                 @7C37:0593
          CALL    editor_prompt "Press function key, or enter"
          CALL    erase_phantom             @7C37:0553
          LEA     goto_xy                   @7C37:028B
          CALL
          CALL    disk_patch                @7C37:0010
                  sector_offset             0 (0h)
          MOV     current_sector_no         0 (0h)
          INT     disk_drive_no             ' ' 0 (00h)
  DISK_PATCH      lines_before_sector       '☻' 2 (02h)
                  header_line_no            ' ' 0 (00h)
 ──Watches─────                                              ──2──

 F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu
```

Figure 26-4: Turbo Debugger's variable window allows us to jump to a procedure.

Once you have the cursor on the line with the CALL WORD PTR [BX] instruction, press F4 and follow that with Shift-F2. You will see Dskpatch draw its screen. Then, you will be returned to Turbo Debugger after you push Shift-F2. This time you won't see any of Dskpatch's screen because Turbo Debugger swapped screens. To flip back to the Dskpatch screen, press the Alt-F5 key. Once you are looking at Dskpatch's screen, pressing any key will return you to Turbo Debugger's screen.

At this point you want to see the value of [BX] so you will know which procedure Dskpatch is about to call. For this, add a watch, which allows you to watch a value. Press Ctrl-W to bring up a dialog box that asks for an expression; type in [BX] and press Enter. You will see a screen like the one in Figure 26-5. As you can see in the Watches window, 8100 is the value you found using Debug with the help of Link's memory map. Here you found the value much more quickly.


≡ File Edit View Run Breakpoints Data Options Window Help READY
Watches
2
F1-Help F2-Bkpt F3-Mod F4-Here F5-Zoom F6-Next F7-Trace F8-Step F9-Run F10-Menu

```
[■]=Module: dispatch File: dispatch.asm 86
        CMP     AL,68                           ;F10--exit?
        JE      END_DISPATCH                    ;Yes, leave
                                                ;Use BX to look through table
        LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
        CMP     BYTE PTR [BX],0                 ;End of table?
        JE      NOT_IN_TABLE                    ;Yes, key was not in the table
        CMP     AL,[BX]                         ;Is it this table entry?
        JE      DISPATCH                        ;Yes, then dispatch
        ADD     BX,3                            ;No, try next entry
        JMP     SPECIAL_LOOP                    ;Check next table entry

DISPATCH:
        INC     BX                              ;Point to address of procedure
►       CALL    WORD PTR [BX]                   ;Call procedure
        JMP     DISPATCH_LOOP                   ;Wait for another key

NOT_IN_TABLE:                                   ;Do nothing, just read next characte
```

Figure 26-5: Turbo Debugger after executing Dskpatch up to the CALL instruction.

Type Alt-X to exit from Turbo Debugger.

Warning

Do not forget to change the DW back to a DB in Dispatch.asm.

You may also want to change back the LINKINFO file. You added the /v switch so Link would add the debugging information to the .EXE file. But this debugging information makes the EXE file quite a bit larger (Dskpatch is about 19K with debugging information, and about 2K without). In any case, you will probably want to remove the /v switch before you give your programs to other people.

Summary

That ends our discussion of debugging techniques. In the next chapter, we will add the procedures to scroll the screen between the two half sectors. Then, in the final part of this book you will learn a number of advanced topics.

By the way, don't forget to fix the bug that you placed in DISPATCH_TABLE.

C H A P T E R

The Other Half Sector

In this chapter you will modify Dskpatch so it can show both halves of the sectors.

File altered: PHANTOM.ASM

Disk file: PHANTO27.ASM

27

Topics Covered

Dskpatch should behave like a word processor when you try to move the cursor below the bottom of the half-sector display—the display should move up one line, with a new line appearing at the bottom. The version of Dskpatch on the disk included with this book does that. In this chapter, you will add skeletal versions of the two procedures, SCROLL_UP and SCROLL_DOWN, that scroll the screen. In the disk version of Dskpatch, SCROLL_UP and SCROLL_DOWN can scroll by any number of lines from 1 to 16 (there are 16 lines in our half-sector display). The versions of SCROLL_UP and SCROLL_DOWN that you will add to Dskpatch here scroll by full half sectors, so you will see either the first or second half of the sector.

Scrolling by Half a Sector

The old versions of PHANTOM_UP and PHANTOM_DOWN restore the cursor to the top or bottom of the half-sector display whenever you try to move the cursor off the top or bottom of the display. You will change PHANTOM_UP and PHANTOM_DOWN so that you can call either SCROLL_UP or SCROLL_DOWN when the cursor moves off the top or bottom of the display. These two new procedures will scroll the display and place the cursor at its new position. The modified versions of PHANTOM_UP and PHANTOM_DOWN (in PHANTOM.ASM) are as follows:

Listing 27-1 Changes to PHANTOM.ASM

```
PHANTOM_UP         PROC
     CALL     ERASE_PHANTOM         ;Erase at current position
     DEC      PHANTOM_CURSOR_Y      ;Move cursor up one line
     JNS      WASNT_AT_TOP          ;Was not at the top, write cursor
~~     MOV      PHANTOM_CURSOR_Y,0    ;Was at the top, so put back there~~
     CALL     SCROLL_DOWN           ;Was at the top, scroll
WASNT_AT_TOP:
     CALL     WRITE_PHANTOM         ;Write the phantom at new position
     RET
PHANTOM_UP      ENDP
PHANTOM_DOWN    PROC
```

```
     CALL       ERASE_PHANTOM              ;Erase at current position
     INC        PHANTOM_CURSOR_Y             ;Move cursor up one line
     CMP        PHANTOM_CURSOR_Y,16            ;Was it at the bottom?
     JB         WASNT_AT_BOTTOM                 ;No, so write phantom
~~     MOV        PHANTOM_CURSOR_Y,15 ;Was at bottom, so put back there~~
     CALL       SCROLL_UP           ;Was at bottom, scroll
WASNT_AT_BOTTOM:                    ;Write the phantom cursor
     CALL       WRITE_PHANTOM
     RET
PHANTOM_DOWN    ENDP
```

Don't forget to change the comment header for PHANTOM_UP and PHANTOM_DOWN to mention that these procedures now use SCROLL_UP and SCROLL_DOWN as follows:

Listing 27-2 Changes to PHANTOM.ASM

```
;-----------------------------------------------------------------------;
; These four procedures move the phantom cursors.                       ;
;                                                                       ;
; Uses:         ERASE_PHANTOM, WRITE_PHANTOM                            ;
;               SCROLL_DOWN, SCROLL_UP                                  ;
; Reads:        PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y                      ;
; Writes:       PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y                      ;
;-----------------------------------------------------------------------;
```

SCROLL_UP and SCROLL_DOWN are both fairly simple procedures because they switch the display to the other half sector. For example, if you are looking at the first half sector, and PHANTOM_DOWN calls SCROLL_UP, you will see the second half sector. SCROLL_UP changes SECTOR_OFFSET to 256; the start of the second half sector moves the cursor to the start of the sector display and writes the half-sector display for the second half. Finally it writes the phantom cursor at the top of this display. You can see all the details for both SCROLL_UP and SCROLL_DOWN in Listing 27-3. These two procedures should be added to PHANTOM.ASM as follows:

Listing 27-3 Procedures added to PHANTOM.ASM (Complete listing in PHANTO27.ASM)

```
        EXTRN     DISP_HALF_SECTOR:PROC, GOTO_XY:PROC
.DATA
        EXTRN     SECTOR_OFFSET:WORD
        EXTRN     LINES_BEFORE_SECTOR:BYTE
.CODE ;-------------------------------------------------------------------;
; These two procedures move between the two half-sector displays.        ;
;                                                                         ;
; Uses:          WRITE_PHANTOM, DISP_HALF_SECTOR, ERASE_PHANTOM, GOTO_XY ;
;                SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR                    ;
; Reads:         LINES_BEFORE_SECTOR                                      ;
; Writes:        SECTOR_OFFSET, PHANTOM_CURSOR_Y                          ;
;-------------------------------------------------------------------------;
SCROLL_UP        PROC
        PUSH     DX
        CALL     ERASE_PHANTOM             ;Remove the phantom cursor
        CALL     SAVE_REAL_CURSOR          ;Save the real cursor position
        XOR      DL,DL                     ;Set cursor for half-sector display
        MOV      DH,LINES_BEFORE_SECTOR
        ADD      DH,2
        CALL     GOTO_XY
        MOV      DX,256   ;Display the second half sector
        MOV      SECTOR_OFFSET,DX
        CALL     DISP_HALF_SECTOR
        CALL     RESTORE_REAL_CURSOR       ;Restore the real cursor position
        MOV      PHANTOM_CURSOR_Y,0        ;Cursor at top of second half sector
        CALL     WRITE_PHANTOM             ;Restore the phantom cursor
        POP      DX
        RET
SCROLL_UP        ENDP
SCROLL_DOWN      PROC
        PUSH     DX
        CALL     ERASE_PHANTOM             ;Remove the phantom cursor
        CALL     SAVE_REAL_CURSOR          ;Save the real cursor position
        XOR      DL,DL                     ;Set cursor for half-sector display
        MOV      DH,LINES_BEFORE_SECTOR
        ADD      DH,2
        CALL     GOTO_XY
        XOR      DX,DX                     ;Display the first half sector
        MOV      SECTOR_OFFSET,DX
        CALL     DISP_HALF_SECTOR
        CALL     RESTORE_REAL_CURSOR       ;Restore the real cursor position
        MOV       PHANTOM_CURSOR_Y,15       ;Cursor at bottom of first half sector
        CALL      WRITE_PHANTOM             ;Restore the phantom cursor
        POP       DX
        RET
SCROLL_DOWN       ENDP
```

SCROLL_UP and SCROLL_DOWN both work nicely, although there is one minor problem with them as Dskpatch stands now. Start Dskpatch and leave the cursor at the top of the screen. Press the cursor-up key and you will see Dskpatch rewrite the first half-sector display because Dskpatch rewrites the screen whenever you try to move the cursor off the top or bottom of the half-sector display.

Here's a challenge for you: Modify Dskpatch so that it checks for two boundary conditions. If the phantom cursor is at the top of the first half-sector display and you press the cursor-up key, Dskpatch should do nothing. If you're at the bottom of the second half-sector display and press the cursor-down key, again Dskpatch should do nothing.

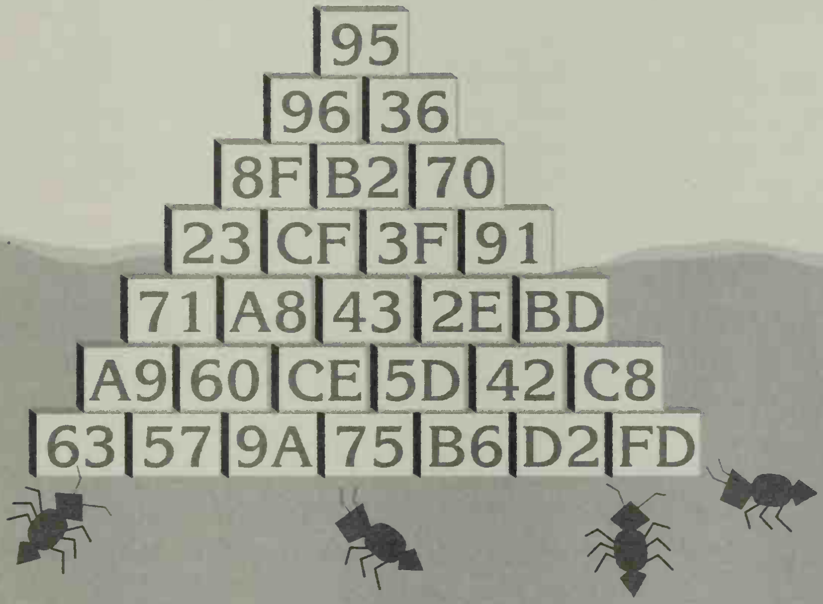
Summary

This chapter ends the coverage of Dskpatch (with the exception of Chapter 30, where we will modify Dskpatch for faster screen writing). Our intent was to use Dskpatch as a "live" example of the evolution of an assembly-language program while providing you with a usable program, and a set of procedures you will find helpful in your own programming. But the Dskpatch you have developed here is not as finished as it could be. You will find more features in the disk version of Dskpatch included with this book. You may find yourself changing that disk version, for "a program is never done . . . but there comes a time when it has to be shipped to users."

This book ends with a number of advanced topics: relocation, writing COM programs, writing directly to the screen, writing C procedures in assembly language, TSR or RAM-resident programs, and protected-mode programming under Microsoft Windows.

P A R T IV

Advanced Topics



CHAPTER

Relocation

In this chapter you will learn how EXE programs are relocated when they are run. You will build a program that accomplishes relocation by itself. We will also introduce you to the old-style segment definitions which you may see in some programs.

28

Topics Covered

Writing COM Programs

Using Full Segment Definitions

Relocation

COM vs EXE Programs

Most of the programs in Parts II and III of this book have been EXE programs with two segments, one for code and one for data. The discussion has glossed over one point in dealing with such programs—relocation. This chapter will cover the relocation process, and the steps DOS takes when it loads an EXE program into memory.

In order to show something of the relocation process, you will build a COM program that does its own relocation (since DOS provides no relocation support for COM programs). Since you haven't dealt with using the assembler to build COM programs yet, we will start with a short look at some new directives that you will need to write COM programs.

Writing COM Programs

Throughout this book you have been using the assembler to build EXE programs, which is what you will probably write most of the time. Some programs, however, need to be COM programs (such as some RAM-resident programs like the one we will write in Chapter 32 and our example program in this chapter). Building COM programs is easy: replace the .MODEL SMALL with .MODEL TINY. The tiny memory model supports .COM programs. You also have to remove the code at the start of your program that sets DS to DGROUP. In a COM program all the segment registers point to the same segment and are set when your program loads.

Using Full Segment Definitions

We will not use the simplified segment definitions (such as .CODE). Instead, we will use the full segment definitions so that everything we do will be out in the open. You will also find it useful to be able to read full segment definitions since some programmers still use them. Full segment directives look very much like procedure definitions, as you can see in this example that defines the code segment.

```
_TEXT   SEGMENT
        .
        .
        .
_TEXT   ENDS
```

Rather than start a code segment with .CODE, you need to bracket the code with a SEGMENT and an ENDS (END Segment) directive. You also have to provide the name of the segment (_TEXT in this example).

In addition to the segment definitions, you need to use another directive called ASSUME. When you are using simplified segment directives, the assembler knows from the .MODEL directive which segments the segment registers will point to. With full segment directives, you need to provide this information to the assembler yourself (since you cannot use the .MODEL directive). For this, you use a new directive, ASSUME, as follows:

```
ASSUME  CS:_TEXT, DS:_DATA, SS:STACK
```

This statement tells the assembler that the CS register will be pointing to your code (which is the case when the program starts to run), that the DS register points to the data segment, and that SS points to the stack segment. The .MODEL directive automatically provides this information to the assembler. (By the way, you will have to set up the last two registers yourself.)

Finally, a COM program, being contained entirely in a single segment, begins with the 256 byte PSP. In order to reserve room for the PSP, COM programs must begin with ORG 100h. The ORG tells the assembler to start the program code at 100h (or 256) bytes into the segment. You will see all of these details in the next section, as well as in Chapter 32.

Relocation

Each of our EXE programs begins with the following code that sets the DS register so it points to the data segment which actually consists of a group of segments called DGROUP.

```
MOV     AX,DGROUP
MOV     DS,AX
```

The question is, where does the value for DGROUP come from? If you think about it, programs can be loaded anywhere into memory. This means that the value of DGROUP won't be known until you know where your program is loaded into memory. As it turns out, DOS performs an operation known as relocation when it loads an EXE program into memory. This relocation process patches numbers such as DGROUP so they reflect the actual location of the program in memory.

To understand this process, you will write a COM program that does its own relocation. The goal is to set the DS register to the beginning of the _DATA segment, and the SS register to the beginning of the STACK segment. This can be accomplished with a bit of trickery. First, you need to ensure that your three segments are loaded into memory in the correct order as follows:

```
Code segment (_TEXT)
Data segment (_DATA)
Stack segment (STACK)
```

Fortunately, we have already taken care of this. When you are using the full segment directives, segments are loaded in the order in which they appear in your source file. A word of warning though: If you ever use the following technique to set segment registers, make sure you know the order in which LINK loads your segments (you can use the .MAP file to check the segment order).

How is the value for DS calculated? Let's begin by looking at the three labels we have placed into various segments in the following listing. Those labels are END_OF_CODE_SEG, END_OF_DATA_SEG, and END_OF_STACK_SEG. They aren't exactly where you might have expected them to be because we define a segment like the following (we need to use full segment definitions for COM programs).

```
_TEXT   SEGMENT
```

This statement does not really tell the linker how to stitch together various segments. So, it starts each new segment on a paragraph boundary—at a hex address that ends with a zero, such as 32C40h. Since the Linker skips to the next paragraph boundary to start each segment, there will be a short, blank area between segments. By placing the label END_OF_CODE_SEG at the beginning of _DATA, you include this blank area. If you had put END_OF_CODE_SEG at the end of _TEXT, you would not include the blank area between segments. (Look at the unassemble listing of the program on page 384. You will see a blank area filled with zeros that is 11 bytes long.)

As for the value of the DS register, _DATA starts at 3AB1:0130, or 3AC4:0000. The instruction OFFSET CGROUP:END_OF_CODE_SEG will return 130h, which is the number of bytes used by _TEXT. Divide this number by 16 to get the number we need to add to DS so that DS points to _DATA. We use the same technique to set SS. The listing for the program, including the relocation instructions needed for a COM file, is as follows:

```
CGROUP  GROUP   _TEXT, _DATA, STACK
        ASSUME  CS:_TEXT, DS:_DATA, SS:STACK


_TEXT   SEGMENT
        ORG     100h                    ;Reserve data area for .COM program
WRITE_sSTRING    PROC    FAR
        MOV     AX,OFFSET CGROUP:END_OF_CODE_SEG
        MOV     CL,4                    ;Calculate number of paragraphs
        SHR     AX,CL                   ; (16 bytes) used by the code segment
        MOV     BX,CS
        ADD     AX,BX                   ;Add CS to this
        MOV     DS,AX                   ;Set the DS register to _DATA

        MOV     AX,OFFSET CGROUP:END_OF_DATA_SEG
        SHR     AX,CL                   ;Calculate paras from CS to stack
        ADD     AX,BX                   ;Add CS to this
        MOV     SS,AX                   ;Set the SS register for STACK
        MOV     AX,OFFSET STACK:END_OF_STACK_SEG
        MOV     SP,AX                   ;Set SP to end of stack area

        MOV     AH,9                    ;Call for string output
        LEA     DX,STRING               ;Load address of string
        INT     21h                     ;Write string

        MOV     AH,4Ch                  ;Ask to Exit back to DOS
        INT     21h                     ;Return to DOS
WRITE_STRING    ENDP

_TEXT   ENDS
```

```
_DATA   SEGMENT
END_OF_CODE_SEG LABEL   BYTE
STRING  DB      "Hello, DOS here.$"
_DATA   ENDS


STACK   SEGMENT
END_OF_DATA_SEG LABEL   BYTE
        DB      10 DUP ('STACK   ')     ;'STACK' followed by three spaces
END_OF_STACK_SEG        LABEL   BYTE
STACK   ENDS


        END     WRITE_STRING
```

Assemble and link this program, just as you would an EXE program, and then type the following to convert writestr.exe into a COM program.

```
EXE2BIN WRITESTR WRITESTR.COM
```

EXE2BIN stands for convert an EXE file into (2) a BINary (COM) file; in other words, EXE to BINary. You can see the results of all this work in the Debug session as follows:

```
C>DEBUG WRITESTR.COM
-U
3AB1:0100 B83001        MOV     AX,0130
3AB1:0103 B104          MOV     CL,04
3AB1:0105 D3E8          SHR     AX,CL
3AB1:0107 8CCB          MOV     BX,CS
3AB1:0109 03C3          ADD     AX,BX
3AB1:010B 8ED8          MOV     DS,AX
3AB1:010D B85001        MOV     AX,0150
3AB1:0110 D3E8          SHR     AX,CL
3AB1:0112 03C3          ADD     AX,BX
3AB1:0114 8ED0          MOV     SS,AX
3AB1:0116 B84600        MOV     AX,0046
3AB1:0119 8BE0          MOV     SP,AX
3AB1:011B B409          MOV     AH,09
3AB1:011D BA0000        MOV     DX,0000
-U
3AB1:0120 CD21          INT     21
3AB1:0122 B44C          MOV     AH,4C
3AB1:0124 CD21          INT     21
3AB1:0126 0000          ADD     [BX+SI],AL
3AB1:0128 0000          ADD     [BX+SI],AL
```

```
3AB1:012A 0000          ADD     [BX+SI],AL
3AB1:012C 0000          ADD     [BX+SI],AL
3AB1:012E 0000          ADD     [BX+SI],AL
3AB1:0130 48            DEC     AX
3AB1:0131 65            DB      65
3AB1:0132 6C            DB      6C
3AB1:0133 6C            DB      6C
3AB1:0134 6F            DB      6F
3AB1:0135 2C20          SUB     AL,20
3AB1:0137 44            INC     SP
3AB1:0138 4F            DEC     DI
3AB1:0139 53            PUSH    BX
3AB1:013A 206865        AND     [BX+SI+65],CH
3AB1:013D 7265          JB      01A4
3AB1:013F 2E            CS:
3AB1:0140 2400          AND     AL,00

-G 120
AX=0946  BX=3AB1  CX=0004  DX=0000  SP=0046  BP=0000  SI=0000  DI=0000
DS=3AC4  ES=3AB1  SS=3AC6  CS=3AB1  IP=0120   NV UP EI PL NZ NA PE NC
3AB1:0120 CD21          INT     21
```

There are a couple of new things in this program. First, you will notice the following line:

```
CGROUP  GROUP   _TEXT, _DATA, STACK
```

You will recall that a group called DGROUP acts as a single segment, but really holds the _DATA, _DATA?, and STACK segments. What you have done here is create your own group, called CGROUP, that contains all the segments in the program. Creating this group enables you to get the offsets of various labels that are in different segments. You want offsets from the start of the entire program, rather than from the start of any one segment. By defining a group called CGROUP, you can get the offset from the start of the program by asking for the offset from the start of this group as follows:

```
MOV     AX,OFFSET CGROUP:END_OF_CODE_SEG
```

In contrast to this state, the statement

```
MOV     AX,OFFSET STACK:END_OF_STACK_SEG
```

sets AX to the offset of END_OF_STACK_SEG from the start of the segment called STACK.

You will almost never need to do this type of relocation yourself since DOS handles this automatically for EXE programs. But it helps to understand what's happening behind the scenes.

COM vs EXE Programs

We will finish this chapter by summarizing the difference between COM and EXE files, and how DOS loads both types of programs into memory.

A COM program stored on disk is essentially a memory image of the program. Because of this, a COM program is restricted to a single segment, unless it does its own relocation, as we did in this chapter.

An EXE program, on the other hand, lets DOS take care of the relocation. This delegating makes it very easy for EXE programs to use multiple segments. For this reason, most large programs are EXE rather than COM programs.

For a final look at COM versus EXE programs, let's examine how DOS loads and starts both of them. This should make the differences between these types of programs clearer and more concrete. We will begin with COM programs.

When DOS loads a COM program into memory, it adheres to the following steps:

- DOS creates the program segment prefix (PSP), which is the 256 byte area you saw in Chapter 11. Among other things, this PSP contains the command line typed.
- DOS next copies the entire COM file from the disk into memory, immediately after the 256 byte PSP.
- DOS then sets the three segment registers DS, ES, and SS to the start of the PSP.
- DOS sets the SP register to the end of the segment, usually FFFE, which is the last word in the segment.
- Finally, DOS jumps to the start of the program, which sets the CS register to the start of the PSP and the IP register to 100h (the start of the COM program).

In contrast, the steps involved in loading an EXE file are somewhat more involved, because DOS does the relocation.

Every EXE file has a header that is stored at the start of the file. This header, or relocation table, is always at least 512 bytes long and contains all the information DOS needs to do the relocation. Microsoft's Macro Assembler includes a program called EXEHDR that you can use to look at some of the information in this header. For example, following is the header we get for the Dskpatch.exe program at the end of Chapter 27:

```
C>EXEHDR DSKPATCH

Microsoft (R) EXE File Header Utility  Version 2.01
Copyright (C) Microsoft Corp 1985-1990. All rights reserved.

.EXE size (bytes)          8fc
Magic number:              5a4d
Bytes on last page:        00fc
Pages in file:             0005
Relocations:               0001
Paragraphs in header:      0020
Extra paragraphs needed:   0241
Extra paragraphs wanted:   ffff
Initial stack location:    0270:0400
Word checksum:             ec2c
Entry point:               0000:0010
Relocation table address:  001e
Memory needed:             11K

C>
```

Near the top of this table, you can see that there is a single relocation entry for the MOV AX,DGROUP instruction. Any time you make a reference to a segment address, as with MOV AX,DGROUP, Link will add a relocation entry to the table. The segment address is not known until DOS loads the program into memory, so you must let DOS supply the segment number.

There are also some other interesting pieces of information in the table. For example, the initial CS:IP and SS:SP values tell you the initial values for Stack and the Entry point. The table also tells DOS how much memory the program needs before it can run (the Memory needed).

Because DOS uses this relocation table to supply absolute addresses for such locations as segment addresses, there are a few extra steps it takes when loading a program into memory. The steps DOS follows in loading an EXE program as follows:

- DOS creates the program-segment prefix (PSP), just as it does for a COM program.
- DOS checks the EXE header to find where the header ends and the program starts. Then it loads the rest of the program into memory after the PSP.
- DOS finds and patches all of the references in the program that need to be relocated, such as references to segment addresses using the header information.
- DOS then sets the ES and DS registers so they point to the start of the PSP. If your program has its own data segment, your program needs to change DS and/or ES so they point to your data segment.
- DOS sets SS:SP according to the information in the EXE header. In the case illustrated, the header states that SS:SP will be placed at 0004:0050. That means DOS will set SP to 0050, and set SS so it is four paragraphs higher in memory than the end of the PSP.
- Finally, DOS jumps to the start of the program using the address provided in the EXE header. This sets the CS register to the start of the code segment, and IP to the offset given in the EXE header.

C H A P T E R

More on Segments and ASSUME

In this chapter you will learn about segment overrides, which are very useful in real programs. You will use them in Chapter 30 to write characters directly to your computer screen. This is much faster than using the ROM BIOS routines to display characters. You will also learn more about ASSUME statements and full segment definitions.

29

Topics Covered

Segment Override

Another Look at ASSUME

Summary

Segment Override

So far you have read and written data located in the data segment. You have been dealing with a single data segment in this book (which is actually several segments grouped into a single segment called DGROUP), so you have not had any reason to read or write data in other segments.

In some cases you will need more than one data segment. A classic example is writing directly to the screen. Most commercial DOS programs write to the screen by moving the data directly into screen memory and, in the interest of speed, completely bypassing the ROM BIOS routines. Screen memory on the PC is located at segment B800h for color adapters (such as EGA and VGA), and at segment B000h for monochrome display adapters (which are not very common these days). You must write in different segments to write directly to the screen.

In this section you will write a short program that shows you how to write to two different segments by using the DS and ES registers to point to the two segments. In fact, many programs that write directly to screen memory use the ES register to point to screen memory.

In this example, you will use full segment definitions to give you more control over segments than the simplified segment definitions. Most of the time you will be able to use the simplified segment definitions. But we chose to use the full segment definitions in this chapter to give you more examples of how to use them. We also used full segment definitions to give you a better understanding of the ASSUME statement that you will need, along with the full segment definitions.

Our program is an EXE program. It is very short, and you can see that it has two data segments along with one variable in each data segment. The program is as follows:

```
DOSSEG

_DATA    SEGMENT
DS_VAR              DW        1
_DATA    ENDS
```

```
EXTRA_SEG       SEGMENT PUBLIC
ES_VAR          DW      2
EXTRA_SEG       ENDS


STACK   SEGMENT STACK
        DB      10 DUP ('STACK   ')     ;'STACK' followed by three spaces
STACK   ENDS


_TEXT   SEGMENT
        ASSUME  CS:_TEXT, DS:_DATA, ES:EXTRA_SEG, SS:STACK

TEST_SEG        PROC
        MOV     AX,_DATA                ;Segment address for _DATA
        MOV     DS,AX                   ;Set up DS register for _DATA
        MOV     AX,EXTRA_SEG            ;Segment address for EXTRA_SEG
        MOV     ES,AX                   ;Set up ES register for EXTRA_SEG

        MOV     AX,DS_VAR               ;Read a variable from data segment
        MOV     BX,ES:ES_VAR            ;Read a variable from extra segment

        MOV     AH,4Ch                  ;Ask to Exit back to DOS
        INT     21h                     ;Return to DOS
TEST_SEG        ENDP

_TEXT   ENDS


        END     TEST_SEG
```

This program will help you learn about segment overrides and the ASSUME directive.

Notice that the data segments and the stack segment come *before* the code segment. We have also put the ASSUME directive after all the segment declarations. As you will see in this section, this arrangement is a direct result of using two data segments. Take a look at the two MOV instructions in the program which are as follows:

```
MOV     AX,DS_VAR
MOV     BX,ES:ES_VAR
```

The ES: in front of ES_VAR in the second instruction tells the 80x86 to use the ES, rather than the DS, register for this operation (to read the data from the extra segment). Every instruction has a default segment register it uses when it refers to data. As with the ES register in this example, you can also tell the 80x86 to use some other segment register for data.

The 80x86 has four special instructions, one for each of the four segment registers. These instructions are the *segment-override* instructions. They tell the 80x86 to use a specific segment register rather than the default, when the instruction following the segment override tries to read or write memory. For example, the instruction MOV AX,ES:ES_VAR is actually encoded as two instructions. You will see the following if you unassemble the test program:

```
2CF4:000D 26            ES:
2CF4:000E 8B1E0000      MOV     BX,[0000]
```

This shows that the assembler translated the instruction into a segment-override instruction, followed by the MOV instruction. Now the MOV instruction will read its data from the ES, rather than the DS, segment. If you trace through this program, you will see that the first MOV instruction sets AX equal to 1 (DS_VAR) and the second MOV sets BX equal to 2 (ES_VAR). In other words, you have read data from two different segments.

Another Look at ASSUME

Let's take a look at what happens when you remove the ES: from the program. Change the line

```
MOV     BX,ES:ES_VAR
```

so it reads:

```
MOV     BX,ES_VAR
```

Because you are no longer telling the assembler that you want to use the ES register when you read from memory, you may incorrectly assume that the assembler will go back to using the default segment (DS). Use Debug to look at the result of this change. You will see the ES: segment override is still in front of the MOV instruction. How did the assembler know that it needed to add the segment override? By using the information in the ASSUME directive, the assembler determined that the variable is in the extra segment, rather than in the data segment.

The ASSUME statement tells the assembler that the DS register points to the segment DATA_SEG, while ES points to EXTRA_SEG. Each time you write an instruction that uses a memory variable, the assembler searches for a declaration of this variable to see which segment it is declared in. Then it searches through the ASSUME list to find out which segment register is pointing to this segment. The assembler uses this segment register when it generates the instruction.

In the case of the MOV BX,ES_VAR instruction, the assembler noticed ES_VAR was in the segment called EXTRA_SEG; the ES register was pointing to that segment, so it generated an ES: segment-override instruction on its own. If you were to move ES_VAR into STACK_SEG, the assembler would generate an SS: segment-override instruction. The assembler automatically generates any segment-override instructions needed, provided that the ASSUME directives reflect the actual contents of the segment registers.

Summary

In this chapter you learned more about segments and how the assembler works with them. You learned about segment overrides, which allow you to read and write data in other segments. You will use such overrides in the next chapter when you write characters directly to the screen. Finally, you learned more about the ASSUME directive.

In the next chapter you will learn how to write directly to screen memory. This will dramatically increase the speed of writing characters to the screen.

C H A P T E R

A Very Fast WRITE_CHAR

In this chapter you will learn about speed and performance. You are going to modify Dskpatch so it displays characters by writing directly to your screen, rather than using the ROM BIOS.

Files altered: DSKPATCH.ASM, KBD_IO.ASM, CURSOR.ASM, VIDEO_IO.ASM

Disk files: DSKPAT30.ASM, KBD_IO.ASM, CURSOR30.ASM, VIDEO_30.ASM

30

Topics Covered

In the beginning of this book we mentioned that many people who write programs in assembly language often do so for speed. Assembly-language programs are almost always faster than programs written in other languages. But you may have noticed that the Dskpatch program does not draw the screen as quickly as many commercial programs. It is slow because we have been using the ROM BIOS routines to display characters on screen. Most programs bypass the ROM BIOS and write characters directly to screen memory in favor of raw speed.

Finding the Screen Segment

Before writing characters directly to screen memory, you need to know where the display memory is and how it stores characters.

Screen memory has its own segment, either B800h or B000h. There are two classes of display adapters: monochrome display adapters and color graphics adapters (CGA, EGA, and VGA). You can have one adapter of each class in your computer at the same time (although not many people do), so IBM gave them non-overlapping screen segments.

Monochrome refers to IBM's monochrome display adapter, Hercules graphics cards, and EGA and VGA cards attached to an IBM monochrome display. Monochrome cards display characters on the screen in green, white, or amber (it depends on the display). They have a limited set of "colors" that can be displayed: normal, bright, inverse, and underlined. Monochrome cards have their screen segment at B000h.

Color graphics adapters can display 16 different text colors at one time, and can be switched to graphics mode. The most common color graphics adapters are EGA and VGA cards, although there are still a few CGA cards from the earlier days. Color graphics adapters have their screen memory at segment B800h.

Users should not need to know which type of display adapter they have. It is up to the program to determine which display adapter is active. For this you can use INT 11h, which returns a list of installed equipment. As you can see in Figure 30-1, bits 4 and 5 tell you if the display is monochrome or color.

The screen segment will be at B000h (monochrome) if both bits are 1, and B800h (color) if otherwise (we will ignore the case when no display adapter is installed).

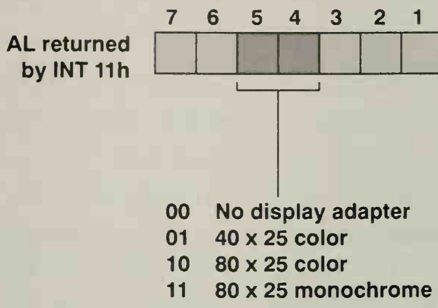


Figure 30-1: The INT 11h equipment flags.

Because you won't know which screen segment to use until you run the program, you will need to call the procedure INIT_WRITE_CHAR, which determines the screen segment, before making any calls to WRITE_CHAR. You will place this call at the start of DISK_PATCH to make sure it is called before any characters are written on the screen. Changes to DSKPATCH.ASM to add this call are as follows:

Listing 30-1 Changes to DSKPATCH.ASM (Complete listing in DSKPAT30.ASM)

```
        EXTRN   WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
        EXTRN   INIT_WRITE_CHAR:PROC
DISK_PATCH      PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        CALL    INIT_WRITE_CHAR
        CALL    CLEAR_SCREEN
        CALL    WRITE_HEADER
                  .
                  .
                  .
```

Next, add INIT_WRITE_CHAR to VIDEO_IO.ASM as follows:

Listing 30-2 Procedure added to VIDEO_IO.ASM

```
        PUBLIC  INIT_WRITE_CHAR
;----------------------------------------------------------------------;
; You need to call this procedure before you call WRITE_CHAR since     ;
; WRITE_CHAR uses information set by this procedure.                   ;
;                                                                      ;
; Writes:       SCREEN_SEG                                             ;
;----------------------------------------------------------------------;
INIT_WRITE_CHAR PROC
        PUSH    AX
        PUSH    BX
        MOV     BX,0B800h                   ;Set for color graphics display
        INT     11h                         ;Get equipment information
        AND     AL,30h                      ;Keep just the video display type
        CMP     AL,30h                      ;Is this a monochrome display adapter?
        JNE     SET_BASE                    ;No, it's color, so use B800
        MOV     BX,0B000h                   ;Yes, it's monochrome, so use B000
SET_BASE:
        MOV     SCREEN_SEG,BX               ;Save the screen segment
        POP     BX
        POP     AX
        RET
INIT_WRITE_CHAR ENDP
```

Note that you are saving the screen segment in SCREEN_SEG (which you will add below). WRITE_CHAR will use this variable when you modify it to write directly to screen memory. Now that you know how to find the screen memory, you will learn how the characters and their attributes are stored.

Writing Directly to Screen Memory

If you were to use Debug to look at screen memory when the first line of the screen is

```
DSKPATCH ASM
```

you would see the following (for a color graphics card):

```
-B800:0
B800:0000  44 07 53 07 4B 07 50 07-41 07 54 07 43 07 48 07   D.S.K.P.A.T.C.H.
B800:0010  20 07 41 07 53 07 4D 07-20 07 20 07 20 07 20 07    .A.S.M. . . . .
        .
        .
        .
```

In other words, there is a 07 between each character on the screen. As you may recall from Chapter 18, 7 is the character attribute for normal text (70h is the attribute for inverse text). Each 7 in the debug display is the attribute for one character, with the character lower in memory. In other words, every character on the screen uses one word of screen memory, with the character code in the lower byte and the attribute in the upper byte. Let's write a new version of WRITE_CHAR that writes characters directly to screen memory. Make changes to VIDEO_IO.ASM as follows:

Listing 30-3 Changes to VIDEO_IO.ASM

```
        PUBLIC  WRITE_CHAR
        EXTRN   CURSOR_RIGHT:PROC
;-------------------------------------------------------------------;
; This procedure outputs a character to the screen by writing directly  ;
; into screen memory, so that characters such as the backspace are      ;
; treated as any other characters and are displayed.                    ;
;                                                                       ;
; This procedure must do a bit of work to update the cursor position.   ;
;                                                                       ;
; On entry:     DL      Byte to print on screen                         ;
;                                                                       ;
; Uses:         CURSOR_RIGHT                                            ;
; Reads:        SCREEN_SEG                                              ;
;-------------------------------------------------------------------;
WRITE_CHAR      PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        PUSH    ES
```

continues

Listing 30-3 continued

```
        MOV     AX,SCREEN_SEG           ;Get segment for screen memory
        MOV     ES,AX                   ;Point ES to screen memory

        PUSH    DX                      ;Save the character to write
        MOV     AH,3                    ;Ask for the cursor position
        XOR     BH,BH                   ;On page 0
        INT     10h                     ;Get row, column
        MOV     AL,DH                   ;Put row into AL
        MOV     BL,80                   ;There are 80 characters per line
        MUL     BL                      ;AX = row * 80
        ADD     AL,DL                   ;Add the column
        ADC     AH,0                    ;Propagate carry into AH
        SHL     AX,1                    ;Convert to byte offset
        MOV     BX,AX                   ;Put byte offset of cursor into BX
        POP     DX                      ;Restore the character

        MOV     DH,7                    ;Use the normal attribute
        MOV     ES:[BX],DX              ;Write character/attribute to screen
        CALL    CURSOR_RIGHT            ;Now move to next cursor position

        POP     ES
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
WRITE_CHAR      ENDP
```

Finally, you need to add a memory variable to VIDEO_IO.ASM:

Listing 30-4 Add DATA_SEG to the start of VIDEO_IO.ASM

```
.MODEL SMALL

.DATA
SCREEN_SEG      DW      0B800h          ;Segment of the screen buffer

.CODE
```

After making these changes, rebuild Dskpatch (you will need to assemble Dskpatch and Video_io) and try the new version. You will see that Dskpatch does not write to the screen any faster than before because you are moving the cursor after you write each character, which slows down the process.

High-Speed Screen Writing

The solution is to rewrite the routines in Video_io and Cursor to keep track of where the cursor should be, instead of moving the cursor (you will move the cursor only when you need to). For this we will introduce two new memory variables: SCREEN_X and SCREEN_Y. This may sound easy, but you will have to change a number of procedures as well as write a few new ones.

There is another optimization you can make. Currently WRITE_CHAR calculates the offset of the cursor into the screen buffer each time you call it. Since you will be keeping track of where the cursor should be, you can also keep track of this offset in the variable SCREEN_PTR.

Listing 30-5 Changes to WRITE_CHAR in VIDEO_IO.ASM

```
; Uses: CURSOR_RIGHT                                             ;
; Reads:          SCREEN_SEG, SCREEN_PTR                         ;
;----------------------------------------------------------------;
WRITE_CHAR        PROC
        PUSH      AX
        PUSH      BX
~~        PUSH      CX~~
        PUSH      DX
        PUSH      ES

        MOV       AX,SCREEN_SEG             ;Get segment for screen memory
        MOV       ES,AX                     ;Point ES to screen memory
        MOV       BX,SCREEN_PTR             ;Pointer to character in screen memory

~~        PUSH      DX                        ;Save the character to write~~
~~        MOV       AH,3                      ;Ask for the cursor position~~
~~        XOR       BH,BH                     ;On page 0~~
~~        INT       10h                       ;Get row, column~~
```

continues

Listing 30-5 continued

```
~~        MOV     AL,DH                         ;Put row into AL~~
~~        MOV     BL,80                         ;There are 80 characters per line~~
~~        MUL     BL                            ;AX = row * 80~~
~~        ADD     AL,DL                         ;Add the column~~
~~        ADC     AH,0                          ;Propagate carry into AH~~
~~        SHL     AX,1                          ;Convert to byte offset~~
~~        MOV     BX,AX                         ;Put byte offset of cursor into BX~~
~~        POP     DX                            ;Restore the character~~

        MOV     DH,7                          ;Use the normal attribute
        MOV     ES:[BX],DX                    ;Write character/attribute to screen
        CALL    CURSOR_RIGHT                  ;Now move to next cursor position

        POP     ES
        POP     DX
~~        POP     CX~~
        POP     BX
        POP     AX
        RET
WRITE_CHAR      ENDP
```

WRITE_CHAR has become quite simple. You must also add three new memory variables to the DATA_SEG in VIDEO_IO.ASM:

Listing 30-6 Changes to .DATA in VIDEO_IO.ASM

```
.DATA
        PUBLIC  SCREEN_PTR
        PUBLIC  SCREEN_X, SCREEN_Y
SCREEN_SEG      DW      0B800h          ;Segment of the screen buffer
SCREEN_PTR      DW      0               ;Offset into screen memory of cursor
SCREEN_X        DB      0               ;Position of the screen cursor
SCREEN_Y        DB      0

.CODE
```

The changes to WRITE_ATTRIBUTE_N_TIMES so it will write directly to the screen are as follows:

Listing 30-7 Changes to WRITE_ATTRIBUTE_N_TIMES in VIDEO_IO.ASM (Complete listing in VIDEO_30.ASM)

```
; Uses:         CURSOR_RIGHT                                           ;
; Reads:        SCREEN_SEG, SCREEN_PTR                                 ;
;----------------------------------------------------------------------;
WRITE_ATTRIBUTE_N_TIMES PROC
        PUSH    AX
~~        PUSH    BX~~
        PUSH    CX
~~        PUSH    DX~~
        PUSH    DI
        PUSH    ES

        MOV     AX,SCREEN_SEG           ;Set ES to point to screen segment
        MOV     ES,AX
        MOV     DI,SCREEN_PTR           ;Character under cursor
        INC     DI                      ;Point to the attribute under cursor
        MOV     AL,DL                   ;Put attribute into AL
ATTR_LOOP:
        STOSB                           ;Save one attribute
        INC     DI                      ;Move to next attribute
        INC     SCREEN_X                ;Move to next column
        LOOP    ATTR_LOOP               ;Write N attributes

        DEC     DI                      ;Point to start of next character
        MOV     SCREEN_PTR,DI           ;Remember where we are

        POP     ES
        POP     DI
~~        POP     DX~~
        POP     CX
~~        POP     BX~~
        POP     AX
        RET
WRITE_ATTRIBUTE_N_TIMES ENDP
```

Most of this procedure should be fairly clear, with the exception of a new instruction: STOSB (*STOre String Byte*). Basically, STOSB is the opposite of the LODSB string instruction that loads a byte from DS:SI and increments the SI register. STOSB stores the byte from AL into the address at ES:DI, then increments DI.

All of the other changes you need to make (with the exception of a simple fix in KBD_IO) are to procedures in CURSOR.ASM. First, you will need to change GOTO_XY so it sets SCREEN_X and SCREEN_Y, and calculates the value of SCREEN_PTR.

Listing 30-8 Changes to GOTO_XY in CURSOR.ASM

```
        PUBLIC  GOTO_XY
.DATA
        EXTRN   SCREEN_PTR:WORD ;Pointer to character under cursor
        EXTRN   SCREEN_X:BYTE, SCREEN_Y:BYTE
.CODE
;----------------------------------------------------------------------;
; This procedure moves the cursor                                      ;
;                                                                      ;
; On entry:     DH      Row (Y)                                        ;
;               DL      Column (X)                                     ;
;----------------------------------------------------------------------;
GOTO_XY         PROC
        PUSH    AX
        PUSH    BX
        MOV     BH,0                    ;Display page 0
        MOV     AH,2                    ;Call for SET CURSOR POSITION
        INT     10h

        MOV     AL,DH                   ;Get the row number
        MOV     BL,80                   ;Multiply by 80 chars per line
        MUL     BL                      ;AX = row * 80
        ADD     AL,DL                   ;Add column
        ADC     AH,0                    ;AX = row * 80 + column
        SHL     AX,1                    ;Convert to a byte offset
        MOV     SCREEN_PTR,AX           ;Save the cursor offset
        MOV     SCREEN_X,DL             ;Save the cursor position
        MOV     SCREEN_Y,DH
        POP     BX
        POP     AX
        RET
GOTO_XY         ENDP
```

As you can see, this listing moves the calculation of the offset to the character under the cursor from WRITE_CHAR, where it was before, to GOTO_XY. You must also modify CURSOR_RIGHT so it updates these memory variables.

Listing 30-9 Changes to CURSOR_RIGHT in CURSOR.ASM

```
        PUBLIC  CURSOR_RIGHT
.DATA
        EXTRN   SCREEN_PTR:WORD ;Pointer to character under cursor
        EXTRN   SCREEN_X:BYTE, SCREEN_Y:BYTE
.CODE
;-------------------------------------------------------------------;
; This procedure moves the cursor one position to the right or to the   ;
; next line if the cursor was at the end of a line.                 ;
;                                                                   ;
; Uses:         SEND_CRLF                                           ;
; Writes:       SCREEN_PTR, SCREEN_X, SCREEN_Y                      ;
;-------------------------------------------------------------------;
CURSOR_RIGHT    PROC
        INC     SCREEN_PTR              ;Move to next character position (word)
        INC     SCREEN_PTR
        INC     SCREEN_X                ;Move to next column
        CMP     SCREEN_X,79             ;Make sure column <= 79
        JBE     OK
        CALL    SEND_CRLF               ;Go to next line
OK:
        RET
CURSOR_RIGHT    ENDP
```

You must also change CLEAR_TO_END_OF_LINE so it uses SCREEN_X and SCREEN_Y rather than the location of the real cursor.

Listing 30-10 Changes to CLEAR_TO_END_OF_LINE in CURSOR.ASM

```
        PUSH    CX
        PUSH    DX
~~        MOV     AH,3                    ;Read current cursor position~~
~~        XOR     BH,BH                   ; on page 0~~
~~        INT     10h                     ;Now have (X,Y) in DL, DH~~
        MOV     DL,SCREEN_X
        MOV     DH,SCREEN_Y
        MOV     AH,6                    ;Set up to clear to end of line
        XOR     AL,AL                   ;Clear window
```

The next few steps require an explanation. Because you are no longer updating the position of the real cursor, the real and virtual cursors will often be out of synchronization. Usually this is not a problem. There are a few cases, however, when you have to synchronize both cursors; sometimes you will want to move the real cursor to where you think the cursor is, and sometimes you will want to move the virtual cursor. For example, before asking the user for input, you need to move the cursor to where you think the cursor should be. You will do this with the procedure UPDATE_REAL_CURSOR, which moves the real cursor.

On the other hand, SEND_CRLF moves the real cursor, so you must call UPDATE_VIRTUAL_CURSOR to move the virtual cursor to where the real cursor is after SEND_CRLF. The two procedures you will need to add to CURSOR.ASM are as follows:

Listing 30-11 Procedures added to CURSOR.ASM

```
        PUBLIC  UPDATE_REAL_CURSOR
;----------------------------------------------------------------------;
; This procedure moves the real cursor to the current virtual cursor   ;
; position.  You'll want to call it just before you wait for keyboard  ;
; input.                                                               ;
;----------------------------------------------------------------------;
UPDATE_REAL_CURSOR      PROC
        PUSH    DX
        MOV     DL,SCREEN_X             ;Get position of the virtual cursor
        MOV     DH,SCREEN_Y
        CALL    GOTO_XY         ;Move real cursor to this position
        POP     DX
        RET
UPDATE_REAL_CURSOR      ENDP


        PUBLIC  UPDATE_VIRTUAL_CURSOR
;-------------------------------------------------------------------;
; This procedure updates the position of our virtual cursor to agree  ;
; with the position of the real cursor.                              ;
;-------------------------------------------------------------------;
UPDATE_VIRTUAL_CURSOR   PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Ask for the cursor position
        XOR     BH,BH                   ;On page 0
        INT     10h                     ;Get cursor position into DH, DL
        CALL    GOTO_XY                 ;Move virtual cursor to this position
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
UPDATE_VIRTUAL_CURSOR   ENDP
```

Note that you are using GOTO_XY to update the three variables SCREEN_X, SCREEN_Y, and SCREEN_PTR.

Finally, you must modify several procedures to use the preceding two procedures. The changes to SEND_CRLF are as follows:

Listing 30-12 Changes to SEND_CRLF in CURSOR.ASM (Complete listing in CURSOR30.ASM)

```
;                                                                   ;
; Uses: UPDATE_VIRTUAL_CURSOR                                       ;
;-------------------------------------------------------------------;
SEND_CRLF       PROC
        PUSH    AX
        PUSH    DX
```

continues

Listing 30-12 continued

```
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
        CALL    UPDATE_VIRTUAL_CURSOR    ;Update position of virtual cursor
        POP     DX
        POP     AX
        RET
SEND_CRLF       ENDP
```

This change makes sure that you know where the cursor is after you have moved the real cursor to the next line.

The changes to READ_STRING that keep the virtual and real cursors in synchronization during keyboard input are as follows:

Listing 30-13 Changes to READ_STRING in KBD_IO.ASM (Complete listing in KBD_IO30.ASM)

```
        EXTRN   UPDATE_REAL_CURSOR:PROC
;-----------------------------------------------------------------------;
;               .                                                        ;
;               .                                                        ;
;               .                                                        ;
; Uses: BACK_SPACE, WRITE_CHAR, UPDATE_REAL_CURSOR                       ;
;-----------------------------------------------------------------------;
READ_STRING     PROC
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                    ;Use SI for index register and

START_OVER:
        MOV     BX,2                     ;BX for offset to start of buffer

READ_LOOP:
        CALL    UPDATE_REAL_CURSOR       ;Move to position of virtual cursor
        CALL    READ_KEY                 ;Read one key from the keyboard
        OR      AH,AH                    ;Is character extended ASCII?
```

Reassemble all three files that you changed (Video_io, Cursor, and Kbd_io) and link Dskpatch. You should notice that screen output is much faster than before.

Summary

Speeding up WRITE_CHAR turned out to be quite a bit of work since you had to change a number of procedures, but the results were well worth the effort. Programs that have snappy screen updates are easier to work with than programs that take longer to paint the screen.

In the next chapter you will learn how to write procedures and functions for the C language in assembly language. For those of you using another language, the next chapter should be a useful starting point.

CHAPTER

Using Assembly Language in C and C++ Programs

In this chapter you will learn how to write assembly-language subroutines for C/C++. You will learn about memory models, and how to write your assembly-language subroutines to work with all memory models. Finally, you will learn how to use in-line assembly directly inside your C/C++ programs.

31

Topics Covered

In this chapter we will show you how to use assembly language in C and C++ programs (we will use C from here on to refer to both C and C++, since C++ is a superset of C) both by writing assembly-language subroutines and using in-line assembly. This will enable you to add assembly-language instructions directly to your C programs. Rather than covering languages like Pascal or BASIC, we are concentrating on C because C is the most popular high-level programming language; most commercial programs are written in C, with a sprinkling of assembly language.

C has become quite popular because it is a modern high-level language that provides many assembly-language type functions, such as the ++ increment operator. Because C is a general-purpose programming language, there are times you will want to write parts of your program in assembly language for speed or low-level access to your machine, etc.

A Clear Screen for C

To assemble the programs in this chapter, you will need Microsoft MASM version 5.1 or later or Borland's Turbo Assembler. We are also using the Microsoft C compiler for the examples in this chapter.

We will start by rewriting a fairly simple procedure, CLEAR_SCREEN, so you can call it directly from C. As you will see, writing assembly language programs for use in C programs is actually quite simple.

The .MODEL directive being used allows you to define the memory model of the program you are building. (We have only used the SMALL memory model in this book.) Starting with version 5.1 of MASM, Microsoft added an extension to the .MODEL directive that allows you to write programs to attach to a number of different languages (including C and Pascal). To tell MASM that you are writing a C procedure, you append a ",C" to the end, as follows:

```
.MODEL  SMALL,C
```

Let's start the rewrite of CLEAR_SCREEN by taking another look at the assembly language version written in Chapter 17 of this book.

```
        PUBLIC  CLEAR_SCREEN
;-------------------------------------------------------------------;
; This procedure clears the entire screen.                          ;
;-------------------------------------------------------------------;
CLEAR_SCREEN    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        XOR     AL,AL                   ;Blank entire window
        XOR     CX,CX                   ;Upper left corner is at (0,0)
        MOV     DH,24                   ;Bottom line of screen is line 24
        MOV     DL,79                   ;Right side is at column 79
        MOV     BH,7                    ;Use normal attribute for blanks
        MOV     AH,6                    ;Call for SCROLL-UP function
        INT     10h                     ;Clear the window
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CLEAR_SCREEN    ENDP
```

This is a fairly simple assembly language procedure. All you have to do to convert this into a C procedure is remove a number of instructions. The new file, shown in Listing 31-1, will hold all of the C procedures written in assembly language in this chapter.

Listing 31-1 The new file CLIB.ASM

```
.MODEL  SMALL,C

.CODE

;-----------------------------------------------------------------------;
; This procedure clears the entire screen.                              ;
;-----------------------------------------------------------------------;
CLEAR_SCREEN    PROC
        XOR     AL,AL                   ;Blank entire window
        XOR     CX,CX                   ;Upper left corner is at (0,0)
        MOV     DH,24                   ;Bottom line of screen is line 24
        MOV     DL,79                   ;Right side is at column 79
        MOV     BH,7                    ;Use normal attribute for blanks
        MOV     AH,6                    ;Call for SCROLL-UP function
        INT     10h                     ;Clear the window
        RET
CLEAR_SCREEN    ENDP

        END
```

(If you are using Turbo Assembler, you will need to add two lines after .MODEL with MASM51 on the first line, and QUIRKS on the second line.) You will notice that we have removed all of the PUSH and POP instructions we used to save and restore registers. These instructions are used in assembly language programs so you do not have to keep track of which registers were changed by procedures you called. This makes programming in assembly language much simpler. C procedures, on the other hand, do not need you to save the AX, BX, CX, or DX registers at all since the C compiler always assumes procedures change these four registers or use them to return values. You are free to use these four registers without saving and restoring them.

Note

You don't need to save and restore the AX, BX, CX, or DX registers in any C procedures you write in assembly language. You do, however, need to save and restore the SI, DI, BP, and segment registers if you change them in your procedures.

Can Change: AX, BX, CX, DX, ES

Must Preserve: SI, DI, BP, SP, CS, DS, SS

Direction flag must be 0 (use CLD)

Following is a very short C program that uses clear_screen(). In fact, that is all this program does.

Listing 31-2 The file test.c

```
main()
{
    clear_screen();
}
```

Use the following steps to assemble CLIB.ASM, compile TEST.C, and link both files together to form TEST.EXE.

```
ML /C CLIB.ASM
CL -C TEST.C
LINK TEST+CLIB,TEST,TEST/MAP;
```

Note

With some linkers or compilers you may have to tell the linker to ignore case since we have created the assembly-language names in uppercase letters, but we're using lowercase letters in the C code. Alternatively, you could write your assembly-language subroutines in lowercase letters to match your C code.

(The CL -C command compiles a file without linking it.) The last line is a bit more complicated than normal because you have asked Link to create a map file so you will know where to find clear_screen() in Debug. Even though TEST.EXE is a fairly small program, the memory map (TEST.MAP) turns out to be rather long because of some extra overhead present in all C programs. Following is an abbreviated version of this map showing the pieces of information we are interested in.

```
                 .
                 .
                 .
 Address         Publics by Name

0054:00EC       STKHQQ
0000:001A       _clear_screen
0054:01D8       _edata
0054:01E0       _end
0054:00DA       _environ
0054:00B3       _errno
0000:01A2       _exit
0000:0010       _main
                 .
                 .
                 .


Program entry point at 0000:002A
```

As you can see, the procedure is actually called _clear_screen instead of clear_screen. Most C compilers put an underscore in front all procedure names. (C compilers also put an underscore in front of variable names.) Using ",C" in the .MODEL directive tells MASM to add an underscore at the front of all assembly-language procedures in the file.

If you don't want a procedure to be public, you can use the PRIVATE keyword to keep a procedure from being declared PUBLIC. For example, this definition creates a private procedure called PrivateProc:

PrivateProc PROC PRIVATE

You will find examples of using the PRIVATE keyword in the library code in Appendix C.

You also may have noticed that we did not include a PUBLIC CLEAR_SCREEN to make CLEAR_SCREEN available to other files. This is another change that ",C" makes for us. The ",C" addition to .MODEL changes the PROC directive so it automatically defines every procedure as a PUBLIC procedure. In other words, if you are writing a C procedure in assembly language (using .MODEL SMALL,C), all of your procedures automatically will be declared PUBLIC for you.

Load TEST.EXE into Debug to see if there are any other changes MASM made for you. Using the address in the load map above (1A, which may be different for your C compiler), the following code is for _clear_screen:

```
C>DEBUG TEST.EXE
-U 1A
4A8A:001A 32C0          XOR     AL,AL
4A8A:001C 33C9          XOR     CX,CX
4A8A:001E B618          MOV     DH,18
4A8A:0020 B24F          MOV     DL,4F
4A8A:0022 B707          MOV     BH,07
4A8A:0024 B406          MOV     AH,06
4A8A:0026 CD10          INT     10
4A8A:0028 C3            RET
      .
      .
      .
```

This is exactly what you have written in CLIB.ASM. In other words, the ",C" at the end of the .MODEL directive only changed the name of the procedure from clear_screen to _clear_screen and declared it as PUBLIC. If this were the only help you got from ",C", we would not be very impressed. Fortunately, there are a number of other areas where MASM helps you to write C procedures in assembly language, specifically in passing parameters to procedures.

Using Clear_screen in C++

C++ adds a slight complication to the picture above as a result of something called *name mangling* or *function-name decorating.* C++ provides extra type-checking for the parameters in subroutines through simple conventions. Names of functions have extra characters added to them that indicate how many and what type of parameters the function defines. For example, if you had a function called

```
some_func(int, double)
```

it might be encoded by your C++ compiler to look something like this:

```
some_func__Fid
```

(In actual practice you will see a different name since different C++ compilers use different schemes for encoding names.) You will notice that the imaginary C++ compiler added __Fid to the end of the name. These extra characters contain information on what some_func refers to. The two underscores separate the subroutine's name from the extra information, which in this example is Fid. The *F* says that some_func is a function, as opposed to a class, and the other letters encode the parameter types, with *i* for the int and *d* for the double.

Each name includes information that uniquely defines the name, as well as the type of all of the parameters in a function. C++ compilers generate such mangled names to ensure that you are using all the correct type of parameters whenever you call an external function.

There are problems, however, with this scheme when you are writing assembly-language subroutines. There is no single standard for name mangling that all compilers use, so you cannot always predict what names a C++ compiler will generate. In addition, you may want to write general-purpose libraries that work both with C and C++ compilers. Since C compilers don't mangle the names, except for adding an underscore at the start of the name, what you need is some way to tell C++ compilers that you want to call an external C function. Fortunately, you can define a function using *extern "C"* to define a function as an external C function. In the example above, you would define some_func as follows in your C++ source file:

If you are trying to use the subroutines in this chapter with a C++ compiler using C++ source files, you will have to declare all of the functions as extern "C". This ensures that your compiler does not mangle the names in your function calls.

```
extern "C" some_func(int, double)
```

You can also define several functions as C functions with the following syntax:

```
extern "C"
    {
    some_func(int, double);
    another_func(int);
    }
```

Passing One Parameter

Throughout this book we have used registers to pass parameters to procedures. This has worked well because we never had more than six parameters (which would require the six registers AX, BX, CX, DX, SI, and DI). C programs, on the other hand, use the stack to pass parameters to procedures. This is where the MASM 5.1 .MODEL extensions really come into play. MASM automatically generates much of the code needed to work with parameters passed on the stack.

To see how this works, we will convert several procedures into C procedures. We will start with a procedure to write a string of characters on the screen. We could convert WRITE_STRING, but because it actually uses a number of other procedures (WRITE_CHAR, CURSOR_RIGHT, INIT_WRITE_CHAR, etc.), we will write a new WRITE_STRING that uses the ROM BIOS to write each character to the screen. This new WRITE_STRING uses INT 10h, function 14 to write each character on the screen. This will not be as fast as WRITE_STRING is now, but it is simple enough so you won't get lost in a lot of code.

Listing 31-3 shows our slow, C version of WRITE_STRING that you should add to CLIB.ASM:

Listing 31-3 Procedure added to CLIB.ASM

```
;-----------------------------------------------------------------;
; This procedure writes a string of characters to the screen.  The     ;
; string must end with          DB      0                              ;
;                                                                      ;
;       void write_string(char *string);                               ;
;-----------------------------------------------------------------;
WRITE_STRING    PROC    USES SI, STRING:PTR BYTE
        PUSHF                           ;Save the direction flag
        CLD                             ;Set direction for increment (forward)
        MOV     SI,STRING               ;Place address into SI for LODSB

STRING_LOOP:
        LODSB                           ;Get a character into the AL register
        OR      AL,AL                   ;Have we found the 0 yet?
        JZ      END_OF_STRING           ;Yes, we are done with the string
        MOV     AH,14                       ;Ask for write character function
        XOR     BH,BH                       ;Write to page 0
        INT     10h                         ;Write one character to the screen
        JMP     STRING_LOOP

END_OF_STRING:
        POPF                                ;Restore direction flag
        RET
WRITE_STRING    ENDP
```

Most of this code should be familiar because it was taken verbatim from our fast WRITE_STRING. One line, however, is quite different. Notice that we have added two pieces of information to the end of the PROC statement.

The first piece, USES SI, tells MASM that you are using the SI register in the procedure. As mentioned above, C procedures must save and restore the SI and DI registers if they modify them. As you will see, the USES SI causes MASM automatically to generate code to save and restore the SI register.

The second piece is used to pass one parameter to the program, which is a pointer to a string, or bytes of characters. STRING:PTR BYTE says that you want to call the parameter STRING, and that it is a pointer (PTR) to a character (BYTE), which is the first character in the string. By giving this parameter a name, you can use the parameter's value by writing its name, as in MOV SI,STRING.

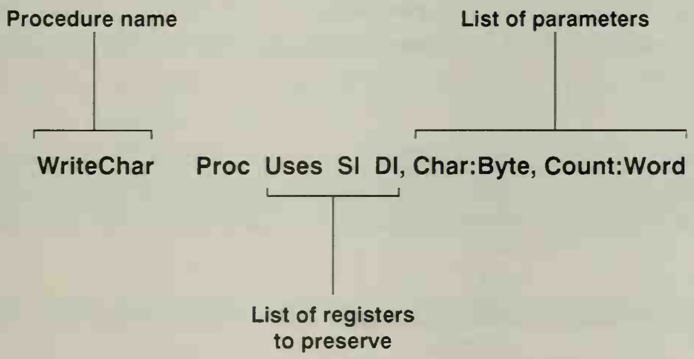

Procedure name
List of parameters
WriteChar Proc Uses SI DI, Char:Byte, Count:Word
List of registers to preserve

Figure 31-1: The Proc directive allows you to define both parameters that are passed on the stack and which registers will be preserved.

The magic of this will become clear as soon as you see the code generated by MASM. Assemble the new CLIB.ASM, then make the following change to TEST.C:

Listing 31-4 Changes to TEST.C

```
main()
{
    clear_screen();
    write_string("This is a string!");
}
```

Recompile TEST.C (with CL -C TEST.C) and link again (with LINK TEST+CLIB,TEST,TEST/MAP;). Looking at the new map file you will see that _write_string is at 33h (you may see a different number depending on the compiler you use).

```
0000:0024       _clear_screen
0056:01EA       _edata
0056:01F0       _end
0056:00DA       _environ
0056:00B3       _errno
0000:01C6       _exit
0000:0010       _main
0000:0033       _write_string
```

Following is the code actually generated by MASM for the write_string we just added to CLIB.ASM (the instructions added by MASM are against a gray background):

```
-U 33
4A8A:0033 55            PUSH    BP
4A8A:0034 8BEC          MOV     BP,SP
4A8A:0036 56            PUSH    SI
4A8A:0037 9C            PUSHF
4A8A:0038 FC            CLD
4A8A:0039 8B7604        MOV     SI,[BP+04]
4A8A:003C AC            LODSB
4A8A:003D 0AC0          OR      AL,AL
4A8A:003F 7408          JZ      0049
4A8A:0041 B40E          MOV     AH,0E
4A8A:0043 32FF          XOR     BH,BH
```

```
4A8A:0045 CD10          INT     10
4A8A:0047 EBF3          JMP     003C
4A8A:0049 9D            POPF
4A8A:004A 5E            POP     SI
4A8A:004B 5D            POP     BP
4A8A:004C C3            RET
```

MASM added quite a few instructions to the ones you wrote. The PUSH SI and POP SI instructions should be clear since we said that MASM would save and restore the SI register in response to USES SI. The other instructions require some explanation.

The BP register is a special-purpose register we have not said much about. If you look at the table of addressing modes in Appendix D you will notice that BP is a little different from other registers in that the default segment for [BP] is the SS register rather than the DS register. This is of interest here because C programs pass parameters on the stack rather than in registers. So the instruction

```
MOV     SI,[BP+04]
```

will always read from the stack, even if SS is not the same as DS or ES (which it often won't be for memory models other than SMALL). Because the BP register is so convenient for working with the stack, C procedures use the BP register to access the parameters passed to them on the stack.

In order to use the BP register, you have to set it to the current value of SP, which the MOV BP,SP instruction does for us. But since the C procedure that called us also uses the BP register to access its parameters, we need to save and restore the BP register. So the assembler automatically generates these instructions (without the comments, of course) that allow us to use the BP register to read parameters from the stack:

```
PUSH    BP                      ;Save the current BP register
MOV     BP,SP                   ;Set BP to point to our parameters
        .
        .
        .
POP     BP                      ;Restore the old value of BP
```

Figure 31-2 shows how the stack would look for a procedure, with two parameters, that uses the SI register. The C call, c_call(param1, param2), pushes the parameters onto the stack, from right to left. By pushing the rightmost parameter first, and the leftmost parameter last, the first parameter will always be closest to the "top of the stack"; in other words, closest to SP. Doing so means that param1 will always have the same offset from BP, no matter how many parameters you actually pass to this subroutine.

The CALL instruction created by the write_string( ) statement pushes the return address onto the stack, at which point our procedure gains control. You will notice at this point that the PUSH SI instruction appears *after* the MOV BP,SP instruction. Once you have set the value of BP, you are free to change the stack as much as you want by PUSHing and POPping registers, and by calling other procedures. Because MASM generates all the needed instructions you do not have to concern yourself with writing these instructions in the correct order.

The first parameter will always be at the same offset from BP, which is 4 for the SMALL memory model (it would be 6 for memory models that require a FAR return address, since a FAR return requires both the old CS and IP values to be on the stack). Looking at the unassembled listing above, you will notice that the assembler translated the MOV SI,STRING instruction into MOV SI,[BP+4], see Figure 31-3. If you had used a memory model with FAR procedures, this would be translated into MOV SI,[BP+6].

C passes parameters on the stack in the opposite order from most other high-level languages. Pascal, BASIC, and FORTRAN, for example, push the first parameter onto the stack first, with the last parameter last, which means the last parameter would be closest to the top of the stack (SP). The offset from BP to the first parameter will depend on the number of parameters you pushed onto the stack. This is not a problem in Pascal, BASIC, or FORTRAN where procedure calls *must* have the same number of parameters as defined in the procedure.

In C procedures, however, you can pass more parameters on the stack than are defined in the procedure. The C printf( ) function is a good example. The number of parameters you pass to printf( ) depends entirely on how many % arguments you have in the string. To allow C procedures to have a variable number of parameters, you need to push the parameters in reverse order so the first parameter will always be closest to SP, and not depend on the number of parameters you actually pushed onto the stack.

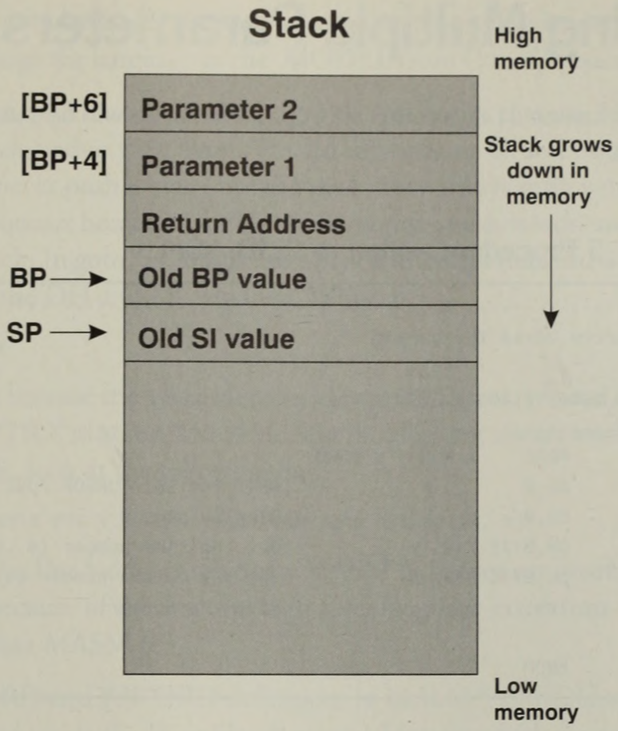


Figure 31-2: How C passes parameters on the stack.

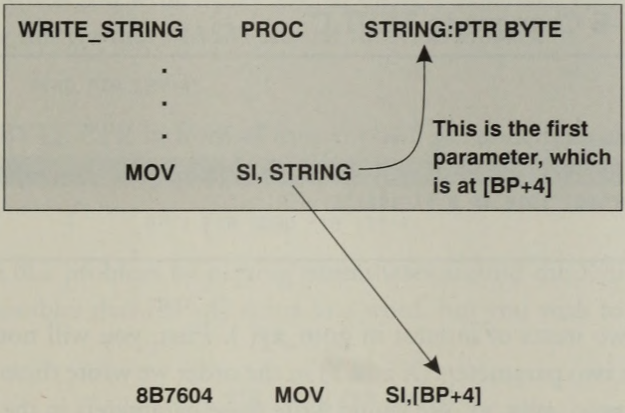

WRITE_STRING PROC STRING:PTR BYTE
.
.
.
MOV SI, STRING
This is the first parameter, which is at [BP+4]
8B7604 MOV SI,[BP+4]

Figure 31-3: The Assembler knows where to find the parameter.

Passing Multiple Parameters

Following is Listing 31-5, another short procedure you will find useful in your C programs.

Listing 31-5 Procedure added to CLIB.ASM.

```
;----------------------------------------------------------------;
; This procedure moves the cursor                                ;
;                                                                ;
;       void goto_xy(int x, int y);                              ;
;----------------------------------------------------------------;
GOTO_XY         PROC    X:WORD, Y:WORD
        MOV     AH,2                    ;Call for SET CURSOR POSITION
        MOV     BH,0                    ;Display page 0
        MOV     DH,BYTE PTR (Y)         ;Get the line number (0..N)
        MOV     DL,BYTE PTR (X)         ;Get the column number (0..79)
        INT     10h                     ;Move the cursor
        RET
GOTO_XY         ENDP
```

Listing 31-6 shows the changes you should make in TEST.C to use goto_xy( ):

Listing 31-6 Changes to TEST.C

```
main()
{
    clear_screen();
    goto_xy(35,10);
    write_string("This is a string!");
}
```

There are two items of interest in goto_xy( ). First, you will notice that we declared the two parameters (X and Y) in the order we wrote them in the procedure call: goto_xy(x, y). We would write these parameters in the same order for a language, like Pascal, which pushes parameters in a different order; MASM handles the differences in order on the stack so you don't have to change your code, or know what order parameters are pushed onto the stack. All you have to do is change the language in the .MODEL from C to, for example, Pascal.

The other change is a bit more subtle. You will notice that we defined X and Y to be words, rather than bytes. We did this because C, and other high-level languages, never push a byte onto the stack, they always push words onto the stack. This occurs because the PUSH instructions push words, and not bytes, onto the stack. In goto_xy, this is not a problem except that you want to move a byte into the DH and DL registers. Writing:

```
MOV     DL,X
```

won't work because the assembler would report an error. Instead, you have to use BYTE PTR X to access X as a byte. But this does not always work in MASM. For example, look at the following line:

```
MOV     DH,BYTE PTR Y          ;Get the line number (0..N)
```

Although this line works correctly in MASM 6, it does not work properly in MASM 5 because of the way the high-level language extensions were implemented before MASM 6.

The X:WORD and Y:WORD definitions in MASM 5's PROC statement are implemented inside the assembler as *macros.* Macros, which we won't cover in this book, are a way to add *features* to the assembler by substituting text for symbols that you type. The parameters X and Y, for example, are actually macros. So when you write MOV DL,X, the X is expanded into the text defined by the macros that MASM created for X, as follows:

```
X       →       WORD PTR [BP+4]
```

If you put BYTE PTR in front of this, you will get something the assembler does not know how to handle, as follows:

```
BYTE PTR X      →       BYTE PTR WORD PTR [BP+4]
```

You can fix this problem by putting parentheses around the X and Y, which tells the assembler that [BP+4] refers to a word, but you wish to treat it as a byte:

```
BYTE PTR (X)      →       BYTE PTR (WORD PTR [BP+4])
```

The parentheses simply tell the assembler to process everything between the parentheses first. In the rest of this chapter, we will write the code specifically for MASM 6.

Returning Function Values

In addition to writing C procedures in assembly language, you will probably want to write C functions in assembly, which is quite simple. C functions return values in the following registers: bytes in AL, words in AX, and long words (two words) in DX:AX, with the low word in AX. If you want to return types with 3 bytes or more than 4 bytes, you will need to consult the *Microsoft Macro Assembler Programmer's Guide*, or the *Turbo Assembler User's Guide* for details.

Here are the registers to use to return values to C programs:

```
Byte        AL
Word        AX
Long        DX:AX
```

The following procedure, which you should add to CLIB.ASM, is a rewrite of READ_KEY that returns the extended key code to C programs.

Listing 31-7 Add this procedure to CLIB.ASM.

```
;-------------------------------------------------------------;
; This procedure reads on key from the keyboard.              ;
;                                                             ;
;       key = read_key();                                     ;
;                                                             ;
; Returns: ASCII code          For all keys that generate characters   ;
;          0x100 + scan code   For special function keys.     ;
;-------------------------------------------------------------;
READ_KEY        PROC
        XOR     AH,AH                   ;Ask for keyboard read function
        INT     16h                     ;Read character/scan code from keyboard
        OR      AL,AL                   ;Is it an extended code?
        JZ      EXTENDED_CODE           ;Yes
NOT_EXTENDED:
        XOR     AH,AH                   ;Return just the ASCII code
        JMP     DONE_READING

EXTENDED_CODE:
        MOV     AL,AH                   ;Put scan code into AL
        MOV     AH,1                    ;Signal extended code
DONE_READING:
        RET
READ_KEY        ENDP
```

Listing 31-8 is a version of test.c that will clear the screen, display a string near the center, and wait until you press the space bar before exiting back to DOS.

Listing 31-8 Changes to TEST.C

```
main()
{
    clear_screen();
    goto_xy(35,10);
    write_string("Press space to continue.");
    while (read_key() != ' ')
        ;
}
```

Using Other Memory Models

All of the assembly subroutines we have shown you were designed for the SMALL memory model. In the SMALL model, your program has a single code segment and a single data segment, which means that all calls are NEAR calls and all the data is in the segment pointed to by DS.

Since most real programs use more than 64K for either code or data, how do you write your assembly-language subroutines to work with these other memory models? Part of the answer is easy, and part of the answer is hard.

The easiest way to demonstrate how to handle the different memory models is to first draw a chart that summarizes the different memory models, which you will find in Table 31-1. This chart shows the default type of function calls and pointers used in the different memory models.

Table 31-1 The different memory models

Model	*Calls*	*Data Pointers*	*DS*
TINY	Near	Near	DGROUP
SMALL	Near	Near	DGROUP
COMPACT	Near	Far	DGROUP
MEDIUM	Far	Near	DGROUP
LARGE	Far	Far	DGROUP
HUGE	Far	Far	DGROUP *

** DS does not point to DGROUP in Borland C++'s HUGE memory model.*

NEAR calls are the same as all the calls you have seen so far. All of the procedures are in a single segment, so all CALLs change the IP register *without* changing the CS register. On the other hand, FAR calls are used when you have more than one segment for the code in your program, which is typically the case for most programs written in C, since they often have more than 64K of code. In this case, a call needs both an offset and a segment value for the function you are calling (which changes both CS:IP). FAR CALLs push both the return offset and segment onto the stack. Along with FAR CALLs the 80x86 also has FAR Return instructions: RETF.

The same types of ideas also apply to the data in your program. Many programs written in C use more than 64K of data, which means they need to use more than one data segment. All programs have a default data segment which is actually a group called DGROUP where your global variables are stored, but this *NEAR data* is limited to 64K in size. Any data beyond this NEAR data must be stored in other data segments. You then access this data by using both an offset and a segment, as you will see below. Table 31-2 summarizes the segments used by many C/C++ compilers (including Microsoft and Borland).

Table 31-2 Segments used with memory models

Model	*Code*	*Data*
TINY	_TEXT	DGROUP (same as _TEXT)
SMALL	_TEXT	DGROUP
COMPACT	_TEXT	DGROUP + other data segments
MEDIUM	*filename*_TEXT	DGROUP
LARGE	*filename*_TEXT	DGROUP + other data segments
HUGE	*filename*_TEXT	DGROUP + other data segments

Note

In most C/C++ compilers, the DS register will always point to the DGROUP data segment, so you can access your NEAR data by using NEAR pointers and references. (NEAR data is any data you define with .DATA or .DATA?.) However, there are cases when this may not be true, such as the HUGE model in Borland's C++ compiler or when you create your own custom memory model. In such cases, you will need to set DS so it points to DGROUP at the start of your assembly-language procedures before you can use any of your NEAR variables. In this chapter we will assume that DS points to DGROUP.

You can see that there are a number of different combinations of NEAR/FAR CALLs and pointers. When you write and assemble a specific library of assembly-language subroutines, it must be built for one of these models. If you build a library for one model, but use it with another, you will almost certainly have problems. For example, if you link some small model assembly-language subroutines with a large model C program, the RET instructions in all your assembly-language code will use NEAR returns, but your C code will make FAR CALLs, which means that the NEAR returns in your assembly-language subroutines won't return properly to your C program since a NEAR return restores the IP register, but *not* the CS register.

What you will need to do is build versions of your libraries for each memory model you want to use. In order to write such libraries, you will have to deal with two separate issues—functions and data references.

Writing NEAR and FAR Functions

Switching between NEAR and FAR subroutines based on which model you are using is trivial. As soon as you change the .MODEL statement in any of your files, MASM will automatically change the RET instruction at the end of your procedures to be either a NEAR or a FAR return, based on the current memory model that you have specified. If you wanted to rewrite all of the functions in CLIB.ASM for the medium model, you would change the first line of CLIB.ASM to read the following:

```
.MODEL  MEDIUM,C
```

Then when you reassemble CLIB.ASM, all the return instructions will be RETF instructions (RETurn Far).

Working with NEAR and FAR Data

Working with FAR data takes more work because you cannot assume, as you have done until now, that the data will be in the segment currently pointed to by the DS register. A FAR pointer contains both an offset *and* a segment since the data could be in any segment. So in order to read a single parameter, you have to use slightly different code than before. For example, the code you have in write_string( ) to retrieve a pointer to STRING in SI (the LODSB instruction retrieves bytes from DS:SI) is as follows:

```
MOV     SI,STRING               ;Place address into SI for LODSB
```

However, in the MEDIUM memory model you need to load both the DS and the SI register, rather than just the SI register. You will use the following form for the MEDIUM model.

```
LDS     SI,STRING               ;Place address into SI for LODSB
```

LDS stands for Load DS, and it loads both the DS register and an index register (SI in this case). To see how this works, let's first look at the machine language generated by this instruction. If you look inside the EXE file at the subroutine write_string, you will find the LDS instruction appears as follows:

```
LDS     SI,[BP+06]
```

As mentioned above, the LDS instruction stands for Load DS because it changes the DS register, as well as an index register. In the instruction above, the name STRING was changed to [BP+06], which is the address of a 2-word pointer on the stack. The first word of this pointer is the offset into the segment, so it is loaded by LDS into the SI register (you can use almost any non-segment register as the second register that LDS uses). The second word, which is the segment value, is loaded by LDS into DS. In other words, LDS SI,STRING loads the FAR pointer represented by STRING into DS:SI.

There is another detail you will need to change. Since LDS changes the value of DS, and since C procedures *must* preserve the value of DS, you will need to change the PROC statement so it will preserve the DS register as follows:

```
WRITE_STRING    PROC    USES SI DS, STRING:PTR BYTE
```

The new USES SI DS tells MASM to generate code to save and restore both SI and DS.

Now we will show you another example that is a little trickier. Say you have a procedure that takes a pointer to an integer that has the following C definition (we will write the procedure in assembly).

```
some_func(int *number)
```

The subroutine in assembly language would then appear as follows for the SMALL memory model.

```
.MODEL  SMALL,C

.CODE

;-------------------------------------------------------------;
; This function modifies the value passed to it.              ;
;                                                             ;
;      some_func(int *number)                                 ;
;-------------------------------------------------------------;
SOME_FUNC       PROC    NUMBER:PTR WORD
        MOV     BX,NUMBER
        MOV     [BX],10
        RET
SOME_FUNC       ENDP
```

Since NUMBER is a pointer to the number, rather than the number itself, you need an extra step. The first MOV instruction moves the address of the actual value into the BX register, and the second MOV moves a number into this address.

In the LARGE memory model this same subroutine appears as follows:

```
.MODEL  LARGE,C

.CODE

;---------------------------------------------------------------;
; This function modifies the value passed to it.                ;
;                                                               ;
;       some_func(int *number)                                  ;
;---------------------------------------------------------------;
SOME_FUNC       PROC    NUMBER:PTR WORD
        LES     BX,NUMBER
        MOV     ES:[BX],10
        RET
SOME_FUNC       ENDP
```

You will notice that in this case we used the LES (Load ES) instruction instead of LDS. LES and LDS work in the same way, except that LES loads ES:*reg* while LDS loads DS:*reg*. C procedures must preserve DS, but they can change ES. Using the ES register means we don't need to include USES DS, which causes an extra PUSH and POP to be created to preserve the value of DS. But you do need to write ES: in front of [BX] so that you will work with a value in the ES, rather than the DS segment.

The last case we will cover is reading from a global variable. When you are dealing with NEAR data, you can simply refer to the name of the variable. For example, to load NUMBER into the AX register, you will use the following instruction:

```
MOV     AX, NUMBER
```

This assumes that the DS register points to the segment that contains NUMBER, but this may not be the case. Most compilers (C as well as other languages) have a default segment that DS always points to, and this segment contains most of your constants and global variables. But if you have more than 64K of constants and global variables, some of the global variables will have to be in segments other than the default segment. In these cases DS won't point to the segment that contains your global variable, so you will have to set ES so it points to the proper segment and use an ES: in front of the address. The code which does this is as follows:

```
MOV     AX, SEG NUMBER              ;Get NUMBER's segment
MOV     ES, AX                      ;Have ES point to this segment
MOV     AX, ES:NUMBER               ;Read the value of NUMBER
```

The SEG directive gives us the segment of a variable. When the assembler encounters this directive, it creates a relocation-table entry for this number since the actual segment number won't be known until DOS loads your program into memory. By the way, this code will work with any memory model, even when you only have a single data segment. However, this code is slower than a single move statement, since you have three move statements.

Note

The MOV ES, AX command takes only 2 clock cycles to run (3 on the 80486), so it is a very quick command. However, this same command is much slower in protected-mode programs, such as in Windows programs. In these cases, the MOV ES, AX command takes 18 clock cycles on the 80386 and 9 cycles on the 80486. As you will see in Chapter 33, this command does a lot of extra work in a protected-mode program. The bottom line is that as long as you are working with a DOS program, the MOV AX, ES command is very fast.

Table 31-3 summarizes the different types of code you need to read different types of C parameters.

Table 31-3 Code to access C parameters

C Parameter	*Small Model Code*	*Large Model Code*
char c	mov al,Byte Ptr c	mov al,Byte Ptr c
int number	mov ax,number	mov ax,number (default data segment)

continues

Table 31-3 continued

C Parameter	*Small Model Code*	*Large Model Code*
int far number	N/A	mov ax, seg number
		mov es, ax
		mov ax, es:number
int *number	**lea** bx,number	**les** bx,number
	mov bx,[bx]	mov bx,**es:**[bx]

Writing General-Model Procedures

All the changes can be rather difficult to keep under control, especially if you need to write procedures that you can use to create code for a number of different memory models. Fortunately, there is a way you can write subroutines that you can reassemble for *any* memory model simply by changing the .MODEL directive at the start of the file. The trick uses something known as *conditional assembly.*

With conditional assembly you can have two versions of code that will be assembled, depending on the outcome of a test. Since you need to generate different code to handle NEAR and FAR pointers, this is a good candidate for conditional assembly. In essence, the kind of code you would write to handle the example SOME_FUNC procedure above is as follows:

```
IF      @DataSize               ;Is this FAR data?
LES     BX,NUMBER               ;Yes, load ES:BX with address
MOV     ES:[BX],10              ;Change value at ES:BX
ELSE
MOV     BX,NUMBER               ;No, load BX with address
MOV     [BX],10                 ;Change value at BX
ENDIF
```

The IF, ELSE, and ENDIF directives allow you to control which code will actually be assembled. In this case, @DataSize is a special MASM value that tells you what kind of pointers you have. When @DataSize is 0, it means you have NEAR pointers; otherwise, you have FAR pointers.

Although this code works very well, it looks much more complicated than it is. Now we will show you a good trick that you can use in your programs. We will use conditional assembly again, but this time we will use it to define some symbols that make it easy to write two lines, rather than seven lines whenever you want to refer to a pointer.

To see what you need to do, notice that the two sets of assembly-language instructions above differ in two places. In the first instruction, you will use LES rather than MOV to load a FAR pointer; in the second instruction, you use ES: in front of [BX] to refer to the value. We will create two symbols that we will use instead of MOV or LES and instead of the ES:. The definitions you will want to use are as follows:

Listing 31-9 The definitions for creating model-independent code.

```
        IF      @DataSize               ;Are pointers FAR?
lodDS   TEXTEQU <LDS>                   ;Yes, use LDS to get DS:pointer
lodES   TEXTEQU <LES>                   ;And use LES to get ES:pointer
refES   TEXTEQU <ES:>                   ;And put ES: in front of refs
        ELSE
lodDS   TEXTEQU <MOV>                   ;No, use MOV to get pointer
lodES   TEXTEQU <MOV>                   ;And also for ES case
refES   TEXTEQU <>                      ;And nothing for the references
        ENDIF
```

You can write the code above as follows, no matter which memory model you are using.

```
lodES   BX,NUMBER               ;Get pointer to the value
MOV     refES [BX],10           ;Change the value
```

This is a lot easier to read, and it generates the correct code based on the current memory model.

The TEXTEQU directive is a directive new to MASM 6. It is very much like the EQU directive you learned in Chapter 14. EQU allows you to assign numeric values to a name; TEXTEQU allows you to assign text values to a name. The brackets (< and >) in a TEXTEQU delimit the text that will be substituted whenever MASM sees the name, such as lodDS, that you have defined.

The final step you might want to take is to use another TEXTEQU macro to actually define the memory model from the ML command line. You can define any macro before MASM starts to assembly your program by using the following syntax:

```
ML /C /DLANG_MODEL=LARGE CLIB.ASM
```

Your .MODEL directive would appear as follows:

```
.MODEL  LANG_MODEL,C
```

To use a different memory model, simply change the LARGE on the ML command line to any of the other names for memory models.

Table 31-4 summarizes the syntax you will use to read parameters using the macros in Listing 31-9.

Table 31-4 Model-independent code to access C parameters

C Parameter	*Model-Independent Code*
char c	mov al, Byte Ptr c
int number	mov ax, number
	(default data segment)
int number	mov ax, seg number
	mov es, ax
	mov ax, es:number
	(any data segment)
int *number	lodES bx, number
	mov bx, refES [bx]

Summary on Writing C/C++ Procedures in Assembly

Since we have covered a lot of material about how to write assembly-language procedures for your C/C++ programs, it is useful to summarize all the rules and steps you will want to use.

1. **Register Usage:** Here is a list of registers that you must preserve and can change freely.

   Preserve: SI, DI, BP, SP, CS, DS, SS

   Can Change: AX, BX, CX, DX, ES

2. **C Parameters:** You can access any procedure parameters directly by name, assuming you have defined them in the PROC statement. The compiler will automatically turn all such references into the form [bp+*x*] for you, which will refer to the parameters stored on the stack, by using the SS segment.

3. **Global C Variables:** When you are working with a memory model that uses NEAR pointers, or when global variables are defined in the default data segment (which is a NEAR data segment), you can refer to any global variable directly by name. The DS register will point to the default data segment, which is a NEAR segment, for most C compilers (except for the Huge model in Borland C++). However, if a global variable is in another segment, you will have to load the ES register by first moving SEG *varname* into another register and then by moving this register into ES.

4. **Far C Pointers/Data:** If you are working with Far pointers or with global variables that are not stored in the default data segment, you will need to set the ES register so it points to your data.

5. **Function Return Values:** Just like in assembly-language subroutines, C functions return values in the registers: AL for Char, AX for Int, DX:AX for Long. For returning pointers, you will need to return an offset in AX for NEAR pointers, and a segment:offset pair in the DX:AX registers for FAR pointers.

Writing In-Line Assembly Code

The final subject we will cover in this chapter is *in-line* assembly. Most C and C++ compilers allow you to write code directly in your C files, rather than requiring you to write separate ASM files to do the work. This can often be the most convenient way of adding the power of assembly language to your C programs.

The syntax you will need to use varies very slightly between Borland's Turbo C++, Microsoft's QuickC, and Microsoft's C++ compilers, but only by some underscore characters. Each compiler has a special keyword that indicates when you "turn on" assembly language mode. For Borland, you use asm; for Microsoft's QuickC compiler you use _asm; for Microsoft C/C++ 7 you use __asm.

The reason for the differences in the syntax of the asm keyword is mostly a result of history. The C language began without any real standards, so different companies added their own private keywords (such as asm and _asm). However, there is now a standards committee that defines standards for the C language and for how C compilers should be written; their standard is known as the ANSI C standard.

The ANSI committee now "allows" compiler writers to add their own, private keywords as long as such names begin with two underscores, as in __asm. For this reason, Microsoft's C/C++ compiler (which is more recent than the QuickC compiler we used in this chapter) uses __asm rather than _asm. As an example, we will rewrite GOTO_XY as a C-language subroutine called GotoXY in Microsoft C:

Listing 31-10 A C version of GOTO_XY using in-line assembly.

```
void GotoXY(
        int x,                          // X coordinate, 0..79
        int y )                         // Y coordinate, 0..??
{
    _asm
        {
        mov     ah,2                    // Use Set Cursor Position call
        mov     dl, Byte Ptr x          // Put X, Y into DL, DH
        mov     dh, Byte Ptr y
        int     10h                         // Call ROM BIOS to move cursor
        }
}
```

You use asm, rather than _asm, in Borland C++; and __asm in Microsoft C/C++ 7. What you will notice about this new subroutine is that it is a mix between C code and assembly-language code. An assembly-language code is inside the the C subroutine, and some C references are inside the assembly-language code. Now let's look at the code actually generated for GotoXY for the LARGE memory model:

```
PUSH    BP
MOV     BP,SP
PUSH    SI
PUSH    DI
MOV     AH,02
MOV     DL,[BP+06]
MOV     DH,[BP+08]
INT     10
POP     DI
POP     SI
MOV     SP,BP
POP     BP
RETF
```

You will notice that the C compiler automatically converted references to the X and Y parameters into forms like [BP+06], which refers to a value on the stack, at offset 6 above BP. This is the same kind of code that MASM would have generated for you, but it is a lot easier to write a little code with in-line assembly in C rather than firing up MASM with a separate source file.

Using Far Data

As long as you are working with a memory model that uses NEAR pointers for all the data (TINY, SMALL, MEDIUM), reading and writing global variables with in-line assembly is very easy: You simply supply the name of the global variable. An example of reading the global variable *x*: is as folows:

```
int     x;                          // Variable in NEAR segment

main()                              // Program shows how to read
{                                   // NEAR data in in-line assembly
    _asm
        {
        mov     ax, x               // Read NEAR data
        }
}
```

In other words, you can reference the variable directly by name, and everything works without any problems because the variable *x* is in the default segment, DGROUP, and the DS register points to DGROUP.

However, when you are dealing with other memory models (COMPACT, LARGE, HUGE) the DS register may not point to the segment that contains your variable. For example, if you have a large array variable, it may be placed in its own data segment, so the code above won't work. Instead, you will need to set a segment register (ES is a good one to use since you are allowed to change its value in C programs) to point to the segment that contains your variable before you can read or write that variable. An example program which shows you how to read the value from x[0] and x[1] is as follows:

```
int     x[30000];                   // Variable in FAR segment

main()                              // Program shows how to read
{                                   // FAR data in in-line assembly
    _asm
        {
        mov     ax, seg x           // Get the segment for x
        mov     es, ax              // Set ES so it points to x
        mov     ax, es:x            // Read x[0]
        mov     bx, es:x[2]         // Read x[1]
        }
}
```

Let's look at this program in detail. The first assembly-language instruction loads the AX register with the number of the segment that contains the array *x*. The next line then sets the ES register so it points to this segment. The last two lines then read two values from this array. Each of these two instructions uses the ES: segment override to read the value from the segment pointed to by ES, rather than the default segment pointed to by DS. We have also shown you how you can directly read elements from this array. The X[2] actually reads the second element of the array. The 2 here is an offset, in *bytes*, from the start of *x*. Since each element of the *x* array is type bytes long (assuming an int variable is 2 bytes), we wrote es:x[2] to read the array element 1.

When you write in-line assembly code that reads either pointers or global variables, you have two choices—either you can write the code so it works just with NEAR or just with FAR pointers, or you can write general-purpose code that sets the ES register before reading or writing variables. If you write code that sets ES, you will be assured that your code will work with any memory model. However, such code will run slower when you are dealing with NEAR pointers and data. If you want the best of both worlds (general-purpose code, but the fastest code), you could write a set of C macros, just like the assembly-language macros in the previous section, to generate the correct code based on the current memory model.

Summary on In-Line Assembly

In general, there are not many restrictions you need to be aware of when you are writing in-line assembly. The more significant ones are as follows (check your compiler's manual for all the details):

1. **Register Usage:** The same rules applied to in-line assembly are applied to writing external assembly-language procedures.

   Preserve: SI, DI, BP, SP, CS, DS, SS

   Can Change: AX, BX, CX, DX, ES

2. **C Parameters:** You can access any procedure parameters directly by name. The compiler will automatically turn all such references into the form [bp+*x*] for you; this will refer to the parameters stored on the stack by using the SS segment.

3. **Local C Variables:** You can access any local C variables directly by name. All local variables are stored on the stack, so the compiler will generate an address of the form [bp+*x*], which always refers to an address on the stack by using the SS segment.

4. **Global C Variables:** You can refer to any global variable directly by name when working with a memory model that uses NEAR pointers

or when global variables are defined in the default data segment (which is a NEAR data segment). However, if a global variable is in another segment, you will have to load the ES register using the method shown above.

5. **Far C Pointers/Data:** If you are working with Far pointers or with global variables that are not stored in the default data segment, you will need to set the ES register so it points to your data.

6. **Function Return Values:** As in assembly-language subroutines, C functions return values in the registers—AL for Char, AX for Int, DX:AX for Long. For returning pointers, you will need to return an offset in AX for NEAR pointers, and a segment:offset pair in the DX:AX registers for FAR pointers.

7. **Comments:** You can use either C or assembly-style comments inside your in-line assembly code. Borland's C++ compiler, however, does *not* allow assembly-style comments using semicolons—you must use C-style comments.

Summary

That wraps up our coverage of using assembly language in C and C++ programs. In Appendix C, and on the disk included with this book, you will find an entire library of assembly-language procedures for writing to the screen, using the mouse, and working with the keyboard, all written using the methods outlined in this chapter. These libraries show how you might put everything together in the real world.

If you want to write procedures for languages other than C or C++, you will need to consult the documentation on your language, or in the assembler that you are using. Not all compilers for the same language (such as Pascal) use the same conventions. So even though MASM (and Turbo Assembler) supports the Pascal conventions, there may be differences if you are not using both an assembler and a compiler from the same company.

The next chapter covers writing RAM-resident programs; the final technical chapter covers protected-mode programming and using assembly-language code in programs for Microsoft Windows.

CHAPTER

DISKLITE, a RAM-Resident Program

In this chapter you will write a very simple RAM-resident program called Disklite. This displays a drive letter in the upper-right corner of your screen whenever any program reads or writes to a disk drive.

Disk file: DISKLITE.ASM

32

Topics Covered

RAM-Resident Programs

Intercepting Interrupts

Disklite

RAM-Resident Programs

RAM-resident programs are almost always written in assembly language to allow maximum access to the ROM BIOS and memory and to keep them small. The Disklite program you will build, for example, weighs in at just 247 bytes. Since RAM-resident programs stay in memory until you restart your computer, and since more programs need 512K or more of memory to run, keeping the size down on RAM-resident programs is very important. If a program is too large, users won't be willing to keep a copy in memory, which is the whole point.

RAM-resident programs usually need to work closely with the ROM BIOS or with your computer's hardware to change how existing functions work, or to add new functions. Disklite, for example, watches the ROM BIOS routines that read from and write to disks so it can display a disk drive "light" on the screen.

Many programmers like to watch the disk drive light during compiles to keep track of the compiler's progress. When a compile takes 30 seconds or a minute, there is not much else you can do. Programmers also like to watch the disk drive light when they are testing programs that read from or write to a disk to see if they are actually accessing the disk. But what happens if you place your computer under your desk? In this case, Disklite provides an on-screen drive light that *lights up* whenever you read to or write from a disk. It also tells you which disk you are accessing.

Intercepting Interrupts

As we mentioned above, Disklite displays the drive light by watching the ROM BIOS routines that read to and write from disk. All disk reading and writing is performed by the INT 13h ROM BIOS routine. DOS uses this service by issuing an INT 13h instruction. Interrupts, as you saw in Chapter 11, use a vector table at the start of memory to determine which routine to call. Each interrupt vector in this table is two words long since it holds the FAR address of the routine that will handle the interrupt. The INT 13h instruction will use the address at 0:4Ch (13h times 4) in memory as the address of the routine that will handle the INT 13h function. In other words, you could change this address to point to your routine instead of the ROM BIOS's routine.

Figure 32-1 shows how INT 13h calls the routine in the ROM BIOS. Now imagine that you changed the interrupt vector to point to your procedure. Then the vector will point to you instead of the ROM BIOS. Now you have taken control of the INT 13h function. But this is not quite what you want. If you completely take over INT 13h, you have to write a program that will do everything INT 13h did, as well as the new functions you want to add. What you should use to do most of the work is the existing ROM BIOS INT 13h routines.

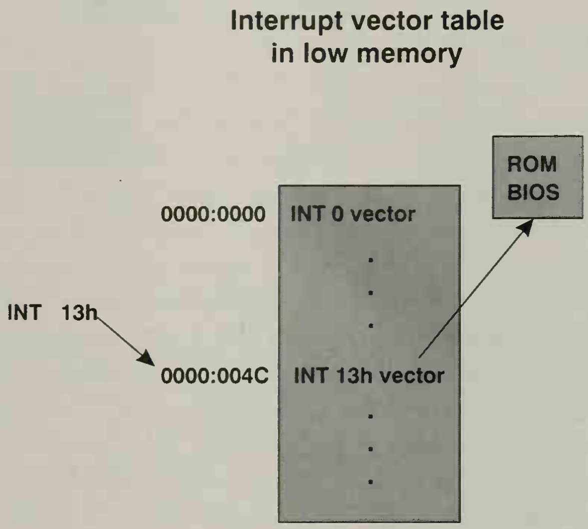


Figure 32-1: INT 13h uses the interrupt vector at 4Ch to determine the address of the routine to call.

Instead of blindly replacing the INT 13h vector, you will save the vector in your own program first, and then you can use the ROM BIOS INT 13h routines by simulating an INT call to the ROM routines. Recall that an INT is like a CALL instruction, but it saves the flags on the stack so they will be restored by an IRET (Interrupt RETurn) instruction. All you need to do is save the address of the INT 13h routines in the variable ROM_DISKETTE_INT, so you can pass control on to the ROM BIOS INT 13h routines with a pair of instructions as follows:

```
PUSHF
CAL         ROM_DISKETTE_INT
```

When the ROM finishes accessing the disk, you will receive control again. This means you can execute some code before, as well as after, you call the ROM's disk functions. This is exactly what you need if you are going to display, then remove, a drive letter. Figure 32-2 shows these steps in more detail.

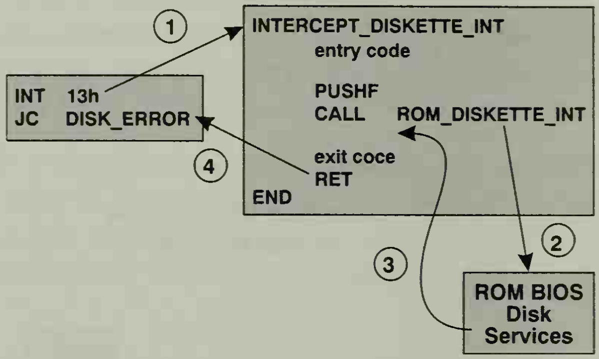


Figure 32-2: Intercepting INT 13h.

The technique we have presented here will work with most ROM BIOS routines. But there is a major caveat. Since DOS is not a multitasking operating system, you cannot make DOS function calls from within an interrupt service routine unless you can be can be absolutely certain DOS was not in the middle of processing a function request. There are ways to ensure this, but they are rather difficult, so we won't cover them in this book. However, you will find some references to this kind of information in the bibliography at the end of the last chapter.

Disklite

Most of the other details of Disklite should either be familiar or documented well enough so you can figure them out. There are a few details, however, that are new or a bit out of the ordinary.

First of all, notice that we are not restoring registers in the procedures of Disklite. Instead, we clearly mark which registers are altered and then we save all the registers that can be altered at the start of INTERCEPT_DISKETTE_INT and nowhere else. We save them only once in order to keep the stack usage to a minimum.

Interrupt service routines generally need to be written so they don't use much of the stack because they are borrowing someone else's stack and there may not be much space left on the stack. You never worried about stack space in your own programs because you gave yourself a large enough stack. You cannot guarantee that everyone will give you a large stack when we get an INT 13h request. For these reasons, many RAM-resident programs set up their own stack.

The procedures GET_DISPLAY_BASE, SAVE_SCREEN, and WRITE_TO_SCREEN should be fairly clear. You have seen GET_DISPLAY_BASE before; the other two should be clear from the last chapter—SAVE_SCREEN saves the two characters in the upper-right corner; WRITE_TO_SCREEN writes two characters in the upper-right corner. WRITE_TO_SCREEN is used both to display the drive letter and to restore the two characters that were on the screen before you displayed the drive letter.

DISPLAY_DRIVE_LETTER is also fairly simple. INT 13h takes a drive number in the DL register. For floppy disk drives, DL will contain 0 for drive A:, 1 for drive B:, and so on. For hard disks, DL starts at 80h. To get the actual drive letter for a hard disk, you subtract 80h and then add the number of floppy disk drives since the first hard disk appears after the last floppy disk.

That leaves you with INIT_VECTORS and GET_NUM_FLOPPIES. INIT_VECTORS shows the details of installing a procedure to intercept an interrupt vector and to keep such a program in memory after you have returned to DOS. First you display an author message, and then you call GET_NUM_FLOPPIES to set NUM_FLOPPIES to the number of floppy disk drives attached to your computer. Next you read and set the INT 13h vector with the INT 21h functions 35h and 25h that read and set interrupt vectors.

You will notice that we put both initialization routines at the very end of Disklite. As it turns out, both these procedures are used only once—when we first load Disklite into memory—so you don't need to keep them in memory after you load Disklite. This is why we have put them at the end. The DOS function call INT 27h, called *Terminate but Stay Resident,* exits your program and keeps most of the program in memory. This function call takes an offset in DX to the first byte you do not want to keep in memory. By setting DX so it points to INIT_VECTORS, you tell DOS to keep all of Disklite in memory *except* for INIT_VECTORS and GET_NUM_FLOPPIES. You could place as much initialization code here as you want without it consuming any memory after Disklite has been installed.

You can enter the following program into a file named DISKLITE.ASM or use the file on disk. Then assemble, link, and convert it into a COM program (you can type ML /AT DISKLITE to create a COM file directly). After you run this program, an inverse X: (where X can be any drive letter) will appear on the very right side of the first line whenever you access a disk drive. To test it, run CHKDSK on any drive.

Listing 32-1 DISKLITE.ASM Program (Disk file DISKLITE.ASM)

```
;-------------------------------------------------------------------;
; Disk Light creates an on-screen version of the disk light that is    ;
; usually on disk drives.  The difference, however, is that this light ;
; will be on only as long as it takes to read or write to the disk.  In;
; other words, it does not stay on while the disk spins without any    ;
; activity.                                                           ;
;                                                                     ;
; This program intercepts the INT 13h vector, which is the entry point ;
; for the ROM BIOS's diskette routine.  On entry, Disklite displays    ;
; the drive letter in the upper-right corner of the screen and        ;
; restores this section of the screen on exit.                        ;
;-------------------------------------------------------------------;


;-------------------------------------------------------------------;
; Here is the DISKLITE's entry point.  It jumps to the initialization  ;
; routine which is at the very end so we can throw it out of memory    ;
; after we've used it.                                                ;
;-------------------------------------------------------------------;
CODE_SEG        SEGMENT
        ASSUME  CS:CODE_SEG, DS:CODE_SEG
        ORG     100h                    ;Reserve for DOS Program Segment Prefix
```

```
BEGIN:  JMP     INIT_VECTORS

AUTHOR_STRING           DB      "Installed Disklite, by John Socha"
                        DB      0Dh, 0Ah, '$'


ROM_DISKETTE_INT        DD      ?

DISPLAY_BASE            DW      ?
OLD_DISPLAY_CHARS       DB      4 DUP (?)
DISPLAY_CHARS           DB      'A', 70h, ':', 70h
NUM_FLOPPIES            DB      ?               ;Number of floppy drives

UPPER_LEFT      EQU     (80 - 2) * 2            ;Offset to drive light

;-------------------------------------------------------------------;
; This procedure intercepts calls to the ROM BIOS's diskette I/O     ;
; vector, and it does several things:                                ;
;                                                                    ;
;       1. Checks to see if the screen is in an 80-column text mode  ;
;          so we can write to the screen.  Disklite won't write any  ;
;          characters to the screen if it's not in an 80-column mode. ;
;       2. Displays the disk drive letter, "A:" for example, in the  ;
;          upper-right corner of the screen.                         ;
;       3. Calls the old ROM BIOS routine to do the actual work.     ;
;       4. Restores the two characters in the upper-right corner of the;
;          screen.                                                   ;
;-------------------------------------------------------------------;
INTERCEPT_DISKETTE_INT  PROC    FAR
        Assume  CS:CODE_SEG, DS:Nothing
        PUSHF                           ;Save the old flags
        PUSH    AX
        PUSH    SI
        PUSH    DI
        PUSH    DS
        PUSH    ES
        CALL    GET_DISPLAY_BASE        ;Calculates the screen's display base
        CALL    SAVE_SCREEN             ;Save two chars in upper right
        CALL    DISPLAY_DRIVE_LETTER    ;Display the drive letter
        POP     ES
        POP     DS
        POP     DI
        POP     SI
```

continues

Listing 32-1 continued

```
        POP     AX
        POPF                            ;Restore the old flags

        PUSHF                           ;Simulate an INT call
        CALL    ROM_DISKETTE_INT        ; to the old ROM BIOS routine

        PUSHF                           ;Save the returned flags
        PUSH    AX
        PUSH    SI
        PUSH    DI
        PUSH    DS
        PUSH    ES
        LEA     SI,OLD_DISPLAY_CHARS    ;Point to the old screen image
        CALL    WRITE_TO_SCREEN         ;Restore two chars in upper right
        POP     ES
        POP     DS
        POP     DI
        POP     SI
        POP     AX
        POPF                            ;Recover the returned flags
        RET     2                       ;Leave the status flags intact
INTERCEPT_DISKETTE_INT  ENDP


;----------------------------------------------------------------------;
; This procedure calculates the segment address for the display adapter;
; that we're using.                                                    ;
;                                                                      ;
; Destroys:     AX                                                     ;
;----------------------------------------------------------------------;
GET_DISPLAY_BASE        PROC    NEAR
        Assume  CS:CODE_SEG, DS:Nothing
        INT     11h                     ;Get the current equipment flag
        AND     AX,30h                  ;Isolate the display flags
        CMP     AX,30h                  ;Is this a monochrome display?
        MOV     AX,0B800h               ;Set for a color graphics adapter
        JNE     DONE_GET_BASE           ;Color graphics, base already set
        MOV     AX,0B000h               ;Set for monochrome display
DONE_GET_BASE:
        MOV     DISPLAY_BASE,AX         ;Save this display base
        RET
GET_DISPLAY_BASE        ENDP
```

```
;-----------------------------------------------------------------;
; This procedure saves the two characters in the upper-right corner of ;
; the screen so that we can restore them later.                   ;
;                                                                 ;
; Destroys:     AX, SI, DI, DS, ES                                ;
;-----------------------------------------------------------------;
SAVE_SCREEN     PROC    NEAR
        Assume  CS:CODE_SEG, DS:Nothing
        MOV     SI,UPPER_LEFT           ;Read chars from the screen
        LEA     DI,OLD_DISPLAY_CHARS    ;Write chars to local memory
        MOV     AX,DISPLAY_BASE         ;Get segment address of screen
        MOV     DS,AX
        MOV     AX,CS                   ;Point to the local data
        MOV     ES,AX
        CLD                             ;Set for auto-increment
        MOVSW                           ;Move two characters
        MOVSW
        RET
SAVE_SCREEN     ENDP


;-----------------------------------------------------------------;
; This procedure displays the drive letter in the upper-right corner of;
; the screen.                                                     ;
;                                                                 ;
; Destroys:     AX, SI                                            ;
;-----------------------------------------------------------------;
DISPLAY_DRIVE_LETTER    PROC    NEAR
        Assume  CS:CODE_SEG, DS:Nothing
        MOV     AL,DL                   ;Get the drive number
        CMP     AL,80h                  ;Is this a hard disk drive?
        JB      DISPLAY_LETTER          ;No, then continue
        SUB     AL,80h                  ;Convert to hard disk number
        ADD     AL,NUM_FLOPPIES         ;Convert to correct disk number
DISPLAY_LETTER:
        ADD     AL,'A'                  ;Convert this into a drive letter
        LEA     SI,DISPLAY_CHARS        ;Point to new char image
        MOV     CS:[SI],AL              ;Save this character
        CALL    WRITE_TO_SCREEN
        RET
DISPLAY_DRIVE_LETTER    ENDP
```

continues

Listing 32-1 continued

```
;-----------------------------------------------------------------;
; This procedure writes two characters in the upper-right corner of the;
; screen.                                                         ;
;                                                                 ;
; On entry:     CS:SI   Screen image for two characters           ;
; Destroys:     AX, SI, DI, DS, ES                                ;
;-----------------------------------------------------------------;
WRITE_TO_SCREEN         PROC    NEAR
        Assume  CS:CODE_SEG, DS:Nothing
        MOV     DI,UPPER_LEFT           ;Write chars to the screen
        MOV     AX,DISPLAY_BASE         ;Get segment address of screen
        MOV     ES,AX
        MOV     AX,CS                   ;Point to the local data
        MOV     DS,AX
        CLD                             ;Set for auto-increment
        MOVSW                           ;Move two characters
        MOVSW
        RET
WRITE_TO_SCREEN         ENDP

;-----------------------------------------------------------------;
; This procedure daisy-chains Disklite onto the diskette I/O vector    ;
; so that we can monitor the disk activity.                       ;
;-----------------------------------------------------------------;
INIT_VECTORS    PROC    NEAR
        Assume  CS:CODE_SEG, DS:CODE_SEG
        LEA     DX,AUTHOR_STRING        ;Print out the author notice
        MOV     AH,9                    ;Display this string
        INT     21h

        CALL    GET_NUM_FLOPPIES        ;See how many floppy drives installed

        MOV     AH,35h                  ;Ask for an interrupt vector
        MOV     AL,13h                  ;Get the vector for INT 13h
        INT     21h                     ;Put vector in ES:BX
        MOV     Word Ptr ROM_DISKETTE_INT,BX
        MOV     Word Ptr ROM_DISKETTE_INT[2],ES

        MOV     AH,25h                  ;Ask to set an interrupt vector
        MOV     AL,13h                  ;Set the INT 13h vector to DS:DX
        MOV     DX,Offset INTERCEPT_DISKETTE_INT
        INT     21h                     ;Set INT 13h to point to our procedure
        MOV     DX,Offset INIT_VECTORS  ;End of resident portion
        INT     27h                     ;Terminate but stay resident
INIT_VECTORS    ENDP


;-----------------------------------------------------------------------;
; This procedure determines how many logical floppy disk drives are in ;
; the system.  The next drive letter will be used for hard disk drives.;
;-----------------------------------------------------------------------;
GET_NUM_FLOPPIES        PROC    NEAR
        Assume  CS:CODE_SEG, DS:CODE_SEG
        INT     11h                     ;Get the equipment flag
        MOV     CL,6
        SHR     AX,CL                   ;Right justify num of floppies
        AND     AL,3                    ;Strip all the other flags
        INC     AL                      ;Returns 0 for 1 floppy
        CMP     AL,1                    ;Is this a one-floppy system?
        JA      DONE_GET_FLOPPIES       ;No, then this is the correct number
        MOV     AL,2                    ;Yes, there are 2 logical drives
DONE_GET_FLOPPIES:
        MOV     NUM_FLOPPIES,AL         ;Save this number
        RET
GET_NUM_FLOPPIES        ENDP


CODE_SEG        ENDS

        END     BEGIN
```

C H A P T E R

Protected-Mode and Windows Programming

In this chapter you will learn the concepts needed to write programs that work in protected-mode environments, such as Microsoft Windows. We will then show you how to write code to test pointers to make sure they are valid, and then show you how to write code to take special advantage of the 80386 instructions in your Windows programs. Please note that you must use MASM 6 to assemble the WINLIB.ASM file in this chapter.

33

Topics Covered

What Is Protected Mode?

Working in Windows

Summary

What Is Protected Mode?

The *protected mode* is a special mode on the 80x86 that is designed to provide protection between different programs running in your computer. As you've seen in the DOS world, any program can read or write to any location in memory. However, in protected-mode programs the operating system can divide memory into a number of separate chunks and the *access rights* for these chunks can be controlled independently for every program in the system. This occurs with the help of the 80x86. As you will see when Windows is discussed, the real world isn't quite as pure as it could be.

A Brief History of the 80x86

Let's begin by looking at how the 80x86 deals with protected-mode memory, which is both very much like what you're used to when using segment registers and very different from what you're used to.

First let's look at the individual microprocessors and what their capabilities and limitations are. The 8088 and 8086 microprocessors used by the original IBM PC don't have any support for a protected mode. They can address one megabyte of memory exactly in the way we've been addressing memory throughout this book. In other words, they use the segmented architecture you have learned in this book.

All other microprocessors in the 80x86 family, starting with the 80286, do support protected-mode programs. The original reason for adding the protected mode to the 80286 was to support more than one megabyte of memory. With the segmented architecture of the 8088 there was no easy way to extend the amount of memory you could work with, short of making all the segment registers longer than 16 bits. Intel could have modified the way programs worked with segments to allow more than one megabyte of memory, but they had other plans.

When Intel began designing the 80286, the IBM PC didn't even exist (the 80286 took almost four years to develop and was introduced in 1982, less than a year after the IBM PC reached the market). As far as most experts believed at the time, the UNIX operating system, not DOS, was the wave of the future.

So Intel designed the 80286 really for two totally separate markets—one was as a faster 8088 for the DOS market, and the other was as a good microprocessor for running UNIX, which meant they needed more than just the ability to manage large amounts of memory. They also needed more advanced protection features to allow programmers to write very robust operating systems like UNIX where a program can crash without causing other programs to crash (Windows 3.1 finally has *some*, but not all, of this protection).

As we all know, DOS took the lead in operating systems, and UNIX never caught on in the 80x86 market place. And as a result, most PC users were not able to take advantage of the additional memory potential on their 80x86 computers until protected-mode operating systems, such as Microsoft Windows, came along.

Addressing Extended Memory

Table 33-1 shows the amount of memory you can use with the current microprocessors in the 80x86 family. As you can see, all of the 80x86 microprocessors, except the 808x, support much more than 1 Mb of memory. To get to this extra memory, also known as *extended memory*, you *must* switch to protected mode. In other words, the protected mode of the 80x86 changes the way the 80x86 addresses memory (and it changes other aspects as well, as you will see in this chapter).

Table 33-1 80x86 Memory Address Size

	8088	*8086*	*80286*	*80386sx**	*80386dx*	*80486*
Memory	1 Mb	1 Mb	16 Mb	16 Mb	4 Gb	4 Gb
Protected Mode	No	No	Yes	Yes	Yes	Yes

** The 80386sx is limited to 16 Mb, rather than 4 Gb. In order to make it easy for computer companies to use the 80386 rather than the 80286. Intel built the 80386sx, which only has 24 address pins, just as the 80286 does.*

The first change we will look at is how the meaning of segment registers changes when you switch to protected mode. In normal mode, which is called *real mode*, you can calculate the address of any area in memory by multiplying the segment by 4 and adding the offset. In protected mode, however, the segment registers actually contain much more information. As you can see in Figure 33-1, the segment registers contain both a visible part, which is 16 bits long, and a much longer hidden part. The hidden part is where all the work of addressing extended memory takes place.

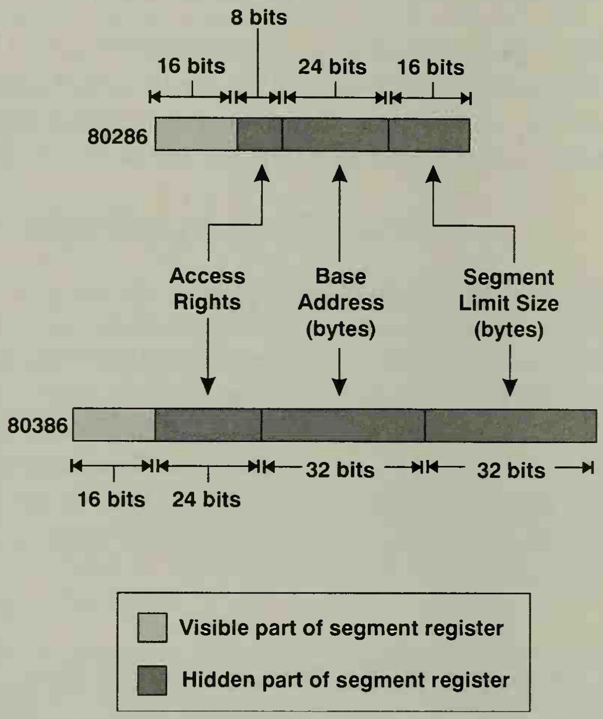


Figure 33-1: In protected mode, the segment registers have a hidden part that contains information on location of a segment, its size, and the access rights.

You will notice that the hidden part of the segment register contains the actual base address of a segment, measured in bytes rather than paragraphs (16 bytes). On the 80286, this field is 24 bits wide, which allows addresses up to 16,772,216, or 16 Mb of memory. The 32-bit field on the 80386 and 80486 allow address up to 4 Gb.

The other two pieces, access rights and the segment limit size, provide some of the protection features we mentioned above. Each segment in protected mode has access rights that control whether you have the right to read from a segment, write to a segment, or run programs in a segment. Segments also have a size, so the 80x86 won't allow you to write or read bytes beyond the end of the segment. If you try to access memory outside these limits, or areas that you don't have the rights to, the 80x86 generates a special type of interrupt known as a *general protection exception.* If you have used Windows 3.1, you have almost certainly heard about and experienced a GPF (general-protection fault), which can be caused by a program attempting to read or write memory that it doesn't own (there are other faults that can also cause a GPF).

The question, then, is how do you load values into these hidden fields? Also, how does the operating system determine which segments you can and cannot access, and where are they located? The answer is that the 80x86 contains one or more tables of such information called descriptor tables.

How Descriptor Tables Work

Whenever you assign a value to a segment register in the protected mode, you are actually using a number, called a *selector*, to look up the actual address and access rights in a table called the *descriptor table* (see Figure 33-2). The descriptor table is a table that maps selector numbers to real addresses in physical memory. The operating system sets up the descriptor table and uses a special protected-mode instruction to tell the 80x86 where to find this table (there are actually two instructions, called LGDT and LLDT, which stand for Load Global/Local Descriptor Table). The hidden part of each segment register is loaded with the values the 80x86 finds in the descriptor table.

The 80x86 allows you to have a number of different descriptor tables. There is a single, *global descriptor table* (GDT), that an operating system generally uses to manage its own private memory. In theory, each program you run should have its own descriptor table, called a *local descriptor table* (LDT). Windows, however, actually uses a single local descriptor table that all programs share, which means Windows programs do not have the kind of protection from each other that they would have if each program had a local descriptor table.

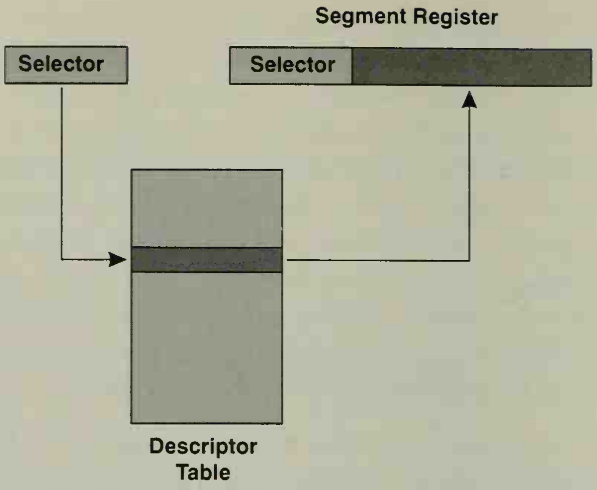


Figure 33-2: When you load a segment register, the 80x86 uses the information in the selector table to load all the hidden fields that tell the 80x86 about the segment.

Each descriptor table has a limit on the number of entries it can contain, which is 8,192. In other words, Windows can never manage more than 8,192 segments at one time. Fortunately, this limit tends not to be a problem for most programs. It does mean, however, that you should be careful not to call GlobalAlloc for many small items since each GlobalAlloc call uses a selector.

Since it takes only 13 bits to address 8,192 selectors, what are the other three bits in the selector used for? It turns out that the lower three bits are used to indicate which descriptor table you should use (local or global) and also privilege information. Since you should never build a selector yourself (you'll always get them from GlobalLock or a similar function), the actual values in these bits aren't very important to you as a programmer.

One interesting side effect of this entire scheme is that Windows, or whatever protected-mode operating system you are using, can move your memory around without your knowledge. Whenever the operating system moves a chunk of memory, all it has to do is update the base address stored in the descriptor table. Your application will continue to use the same selector as if it were a segment, and everything works.

Working in Windows

Let's take a look at how this knowledge, as well as your knowledge of assembly language, can help you write better Windows programs. In this section we will show you some tricks that are very simple using in-line assembly that would be very difficult, if not impossible, to write using straight C code. We will assume here that you have written Windows programs in C, so we won't explain any of the Windows concepts that would be familiar to anyone who has written Windows programs in C.

For our first example, we will show you how to verify that a C-style far pointer (2 words, with a segment and an offset) is legal. As prudent programmers, there are times when it would be nice to test a pointer to see if it is valid before using it. In the DOS world this is typically done by looking at the values referred to by the pointer to see if they are valid. However, you cannot do this in protected mode since you are likely to cause a GPF if the selector is not valid, or if the segment is not large enough. Unfortunately, you cannot protect yourself against these types of problems without some assembly language code. (Windows 3.1 does have a set of functions whose names start with IsBad that you can use to test pointers, but these functions are not available in Windows 3.0.)

Validating Pointers, Part I

Testing a pointer to see if it is valid is easy by using a small amount of assembly language. There are some assembly-language instructions available on the 80x86 that do all the work for you. The VERR and VERW instructions (VERify Read and VERify Write) allow you to test a selector to see if you have read and write privileges. The LSL instruction allows you to retrieve the size of a segment from the descriptor table.

Let's start by looking at the VERR instruction to see how it works. We will use a very short C program to test VERR. This small test program uses a feature you will find in both the Microsoft and the Borland C compilers, which allows you to write simple Windows programs that use the printf( ) function to display output in a window. Microsoft calls this feature QuickWin and Borland calls it EasyWin. The test programs in this section all use printf( ) to display output. Check your documentation on how to use these libraries. In the case of QuickC for Windows, which we used in this chapter, select Project... from the Options menu, then select the QuickWin EXE option in this dialog box.

The VERR instruction uses a single parameter, which is the value of a selector. If the selector is a valid selector and you have read rights to this selector, the VERR instruction sets the zero flag; otherwise it clears the zero flag. Far pointers are always 2 words long in Windows, with the selector in the upper word and the offset in the lower word. The following program checks the selector to see if it is valid:

```
#include        <windows.h>
#include        <stdio.h>

main()
{
    BYTE huge *  lpNum;
    int  valid;                         // -1 if the pointer is valid


    lpNum = (LPBYTE) (5 * 65536);       // This is an invalid selector
    valid = -1;                         // Set to invalid initially

    _asm {
        verr    Word Ptr lpNum[2]       // Is selector valid for reading?
        jz      doneCheck1              // Yes, then we're all done
        mov     valid, 0                // No, report not valid
        }
doneCheck1:

    if (valid)
        printf("Pointer is valid\n");
    else
        printf("Invalid pointer\n");

    return 0;
}
```

This program is not the most elegant. The problem is that the VERR instruction sets or clears the zero flag, but C programs work with variables in memory, rather than flags. So you have to use the variable *valid* to report whether a selector was valid or not, which you can do by setting *valid* to –1 before the VERR instruction. If VERR sets the zero flag, the JZ instruction skips over the instruction that clears *valid.*

You will notice that we had to use a label in the C code to make this test work. If you wanted to check a number of pointers to see if they were valid, you wouldn't want to add this code, with a different label, each time you wanted to test a pointer. Instead, you will probably want to write a function call ValidPtr that tests a pointer to see if it is valid. Such a subroutine appears as follows:

```
//------------------------------------------------------------------//
// This function tests a pointer first to see if the selector part is   //
// valid.                                                               //
//                                                                      //
// Returns:      0      The pointer isn't valid                         //
//              -1      The pointer is valid                            //
//------------------------------------------------------------------//
BOOL ValidPtr(
    void FAR *lp )                      // Pointer you want to test
{
    _asm
        {
        mov     ax, 0                   // Return false by default
        verr    Word Ptr lp[2]          // Is selector valid for reading?
        jnz     doneTest                // No, report not valid

        not     ax                      // Pointer is valid, return -1
        }
doneTest:
    ;
}
```

This function does exactly what the in-line code in our previous example did, except that it's now a function. You will notice that this function returns a value in the AX register. We mentioned in Chapter 31 that C functions return word values in the AX register.

Determining the Size of Segments

We are not finished yet, though. The test you have completed so far tested only the selector part of a far pointer. As long as the selector is invalid, the ValidPtr function correctly reports that the pointer is not valid. But what happens if the selector is valid but the offset points past the end of the segment? In this case we need to actually check the size of the segment, which we do using the LSL instruction (Load Segment Limit). This instruction looks up the size of a segment in the descriptor table and returns the size. For example, if you have a selector in the SI register, the following instruction will return the address of the last byte in the segment that you can address:

```
LSL     AX, SI
```

Adding one to this number will give you the size of the segment referred to by SI. By putting this all together, we can write a new version of our test program that tests a selector to see if it is valid, and if it is, it reports the size of the segment. A new version of this test program follows:

```
#include        <windows.h>
#include        <stdio.h>

//-----------------------------------------------------------------//
// This function tests a pointer first to see if the selector part is   //
// valid.                                                          //
//                                                                 //
// Returns:      0      The pointer isn't valid                    //
//              -1      The pointer is valid                       //
//-----------------------------------------------------------------//
BOOL ValidPtr(
    void FAR *lp,                       // Pointer you want to test
    WORD cwSize )                       // Size of object at *pointer
{
    _asm
        {
        mov     ax, 0                   // Return false by default
        verr    Word Ptr lp[2]          // Is this valid for reading?
        jnz     doneTest                // No, report not valid

        not     ax                      // Pointer is valid, return -1
        }
doneTest:
    ;
}


//-----------------------------------------------------------------//
// This function returns the size of a segment pointed to by a far      //
// pointer.                                                        //
//                                                                 //
// Returns:     size    The size, in bytes, of a segment           //
//-----------------------------------------------------------------//
```

```
DWORD SegSize(
    void FAR *lp )                          // Far pointer to some memory
{
    DWORD i = 0;                            // Size of the segment

    _asm
        {
        lsl     ax, Word Ptr lp[2]          // Get size of segment into ax
        mov     Word Ptr i, ax              // Save size in i
        }
    i++;                                    // Convert from last byte to size
    return i;
}


main()
{
    HANDLE hmem;
    BYTE huge *  lpNum;
    int  valid;                             // -1 if the pointer is valid


    hmem = GlobalAlloc(GMEM_MOVEABLE, 10L);
    lpNum = (LPBYTE) GlobalLock(hmem);

    if (ValidPtr(lpNum))
        printf("Pointer is valid\nSize = %ld", SegSize(lpNum));
    else
        printf("Invalid pointer\n");

    GlobalUnlock(hmem);
    GlobalFree(hmem);

    return 0;
}
```

When you run this program, it should display the following:

```
Pointer is valid
Size = 32
```

The size returned by this function is actually a little larger than the chunk of memory you asked for (10 bytes) because Windows always returns chunks of memory that are a multiple of 32 bytes in size.

There is a problem with this program, however. Change the size of the segment you allocate from 10L to 66000L.

```
hmem = GlobalAlloc(GMEM_MOVEABLE, 66000L);
```

Then run this program in 386-Enhanced mode. What you will discover is that SegSize( ) now returns the wrong size.

```
Pointer is valid
Size = 480
```

To understand why, we first need to look at how Windows normally works with chunks of data larger than 64 Kb.

Whenever you work with chunks of memory larger than 64 Kb in Windows, you need to use huge pointers. Huge pointers are like far pointers because they contain both a segment and an offset, but unlike far pointers, they can work with data objects larger than 64 Kb. Whenever you reference memory using a huge pointer, it loads the ES register with the segment (selector) part of the pointer and it usually loads the offset into the BX register, so the code to read a word from a far pointer might appear as follows:

```
LES     BX, farPtr
MOV     AX, ES:[BX]
```

The interesting part, however, is the code that handles array indexes or adding a number to a pointer. When you are dealing with a far pointer, these operations simply change the value of the offset. But with huge pointers, you may need to change *both* the offset *and* the segment (selector) value. In other words, memory chunks larger than 64 Kb are actually allocated as more than one selector. For example, you would need 4 selectors for 200,000 bytes of memory (3 * 65536 = 196,608, so you need 4 selectors for 200,000 bytes). So when you call GlobalAlloc to allocate a 200,000 byte chunk of memory, Windows creates 4 contiguous selectors to access all 200,000 bytes of memory. The first selector gives you access to the first 64 Kb of memory, the next selector to the next 64 Kb chunk of memory, and so on. Whenever you cross a segment boundary, your C/C++ compiler effectively adds 8 to the selector (it uses 8 rather than 1 because the lower three bits should not be changed since they are reserved for other uses).

The problem in 386-Enhanced mode is that Windows makes a couple of small changes to this model. Whenever you are in 386-Enhanced mode Windows knows that the 80386 and above can work with segments larger than 64 Kb (segments can be up to 4 Gb). So when you allocate 200,000 bytes of memory, Windows creates 4 contiguous selectors, just as you and your C/C++ compiler would expect. But the first selector will actually have a limited size of 199,999 (the last addressable byte in the segment). The second selector likewise will have a limited size of 199,999 – 65536 = 134,463. In other words, each selector's limited size will reflect the last byte at the end of the large chunk of memory that you allocated (see Figure 33-3).

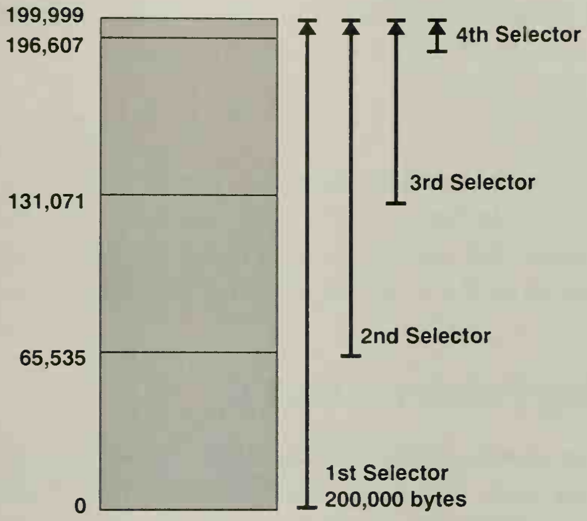


Figure 33-3: This figure shows you how the descriptors created for a large GlobalAlloc cover the memory allocated.

How do you get the actual size of a segment on a 80386 or better processor? For that answer you have to turn back to MASM since you have to use some assembly-language instructions that aren't available in the in-line assembler in most C/C++ compilers.

The 32-Bit Registers

All of the registers we have used in this entire book have been 16 bits long. The 80386 and above processors actually have 32-bit registers that you can use that are longer versions of the 16-bit registers. So just as AX is the 16-bit version of AL, EAX is the 32-bit version of AX (the E stands for Extended). As long as you are writing a program to run on at least the 80386, you can add an E in front of any of the 80x86 registers we've been using. For example, you can add 1 to a 32-bit number by using the following instruction:

```
ADD     EAX, 1
```

To use any of these extended registers in your assembly-language programs you have to first tell the assembler that it can use 80386 instructions (normally the assembler allows only 8088 instructions to ensure that your programs will run on any DOS computer). You turn on 80386 instructions using the .386 directive after the .MODEL directive in your programs:

```
.MODEL  SMALL,C
.386
```

Now you can use the extended registers to get the correct size of a segment larger than 64 Kb. In the next section you will write a set of subroutines in assembly language that you can use to get the correct size of a segment, and to check a pointer to see if it's valid, by checking both the selector and the offset.

Validating Pointers, Part II

In this section we will present three functions that work correctly in all cases when running under Windows. We have written the functions SegSize, ValidReadPtr, and ValidWritePtr as you can see in Listing 33-2. The following listing provides a sample program that both shows you how to use these functions and also shows you the segment sizes for the selectors you get when you allocate objects larger than 64 Kb. Following is the output from Listing 33-1.

```
Selector 0 size = 200000
Selector 1 size = 134464
Selector 2 size = 68928
Selector 3 size = 3392
Invalid pointer
```

When you run the program in 386-Enhanced mode, each selector's size reflects the amount of space to the end of the object you allocated. This is very significant because you can write assembly-language subroutines and functions that work with large chunks of data (larger than 64 Kb) working with just a single segment. You can do this for two reasons: first, Windows sets up the first selector so you can address *all* of the memory you allocated with a single GlobalAlloc call; and second, your assembly-language programs can use the EBX register with 32-bit offsets instead of the usual 16-bit offsets you have used so far. To assemble the WINLIB.ASM file in Listing 33-2, you must use MASM 6.

Listing 33-1 WINTEST.C: A sample program to demonstrate using the SegSize and ValidReadPtr functions

```
#include <windows.h>
#include <stdio.h>

BOOL  ValidReadPtr(void FAR *lp, WORD cwSize);
DWORD SegSize(void FAR *lp);

main()
{
    HANDLE hmem;
    BYTE huge *  lpNum;
    DWORD size;
    int i;


    hmem = GlobalAlloc(GMEM_MOVEABLE, 200000L);
    lpNum = (LPBYTE) GlobalLock(hmem);

    for (i = 0; i < 4; i++)
         {
         size = SegSize(lpNum);
         printf("Selector %d size = %ld\n", i, SegSize(lpNum));
         lpNum += 65536L;
         }

    if (ValidReadPtr(lpNum, sizeof(BYTE)))
         printf("Pointer is valid\nSize = %ld", SegSize(lpNum));
    else
         printf("Invalid pointer\n");

    GlobalUnlock(hmem);
    GlobalFree(hmem);

    return 0;
}
```

Listing 33-2 WINLIB.ASM: The functions to test a pointer to see if it is valid, and to get the size of a segment

```
;-----------------------------------------------------------------------;
; This file contains the following functions that you can call from    ;
; C:                                                                    ;
;                                                                       ;
;       DWORD SegSize(void FAR *lp);                                    ;
;       BOOL  ValidReadPtr(void FAR *lp, WORD cwSize);                  ;
;       BOOL  ValidWritePtr(void FAR *lp, WORD cwSize);                 ;
;-----------------------------------------------------------------------;

.MODEL  SMALL,C
.386

.CODE

;-----------------------------------------------------------------------;
; This function tests to see if the processor is an 80286, as opposed  ;
; to an 80386 or above.                                                 ;
;                                                                       ;
; The 80286 processor always clears the upper four bits when it        ;
; transfers a word from the stack to the status register, while the    ;
; 80386 and above do not, which is how we can tell when we have an     ;
; 80286.                                                                ;
;                                                                       ;
; Returns:      ZR      Processor is an 80286 if zero flag set          ;
;               NZ      80386 or above                                  ;
;-----------------------------------------------------------------------;
Is286   proc    private
        pushf                           ; Put flags onto the stack
        pop     ax                      ; And then into AX
        or      ah,0F0h                 ; Set high 4 bits of flags
        push    ax                      ; Put back onto stack
        popf                            ; And back into flag register
        pushf                           ; Then put back into ax
        pop     ax
        and     ah,0F0h                 ; Report if upper bits zero
        ret
Is286   endp
```

```
;--------------------------------------------------------------------;
; This function returns the size of the segment whose selector is in   ;
; the upper word of p:                                               ;
;                                                                    ;
;       long    SegSize(void FAR *p);                                ;
;--------------------------------------------------------------------;
SegSize proc    p:Ptr
        call    Is286                   ; Is this a 286 processor?
        je      SegSize286              ; Yes, use 16-bit register version

SegSize386:
        movzx   ebx, Word Ptr p[2]      ; Get the selector into ebx
        lsl     eax, ebx                ; Get last offset in seg into eax
        inc     eax                     ; Add 1 to convert to segment size
        mov     edx, eax                ; Lower word in ax, full in edx
        shr     edx, 16                 ; Move upper word into dx
        jmp     doneSegSize             ; We're all done

SegSize286:
        xor     dx, dx                  ; Set upper word to 0
        mov     bx, Word Ptr p[2]       ; Get the selector into bx
        lsl     ax, bx                  ; Get last offset in seg into ax
        add     ax, 1                   ; Add 1 to convert to segment size
        adc     dx, 0                   ; Propagate carry to dx

doneSegSize:
        ret
SegSize endp


;--------------------------------------------------------------------;
; This function is a private function that checks an offset to see if  ;
; it is inside the allowed range.                                    ;
;                                                                    ;
; Returns:      AX       0      Offset isn't valid                   ;
;                       -1      Valid offset                         ;
;--------------------------------------------------------------------;
ValidOffset     proc    private lp:Ptr, cwSize:Word
        INVOKE  SegSize, lp
        sub     ax, cwSize              ; Subtract data size from seg size
        sbb     dx, 0                   ; Propagate borrow
        or      dx, dx                  ; Is segment > 64 K?
        jmp     IsValidOffset           ; Yes, valid since offset always <= 64K
```

continues

Listing 33-2 continued

```
         cmp    Word Ptr lp, ax        ; Is offset too high?
        ja      InvalidOffset          ; Yes, pointer isn't valid

IsValidOffset:
        xor     ax, ax
        not     ax                     ; Return TRUE
        jmp     DoneValidOffset        ; We're all done

InvalidOffset:
        xor     ax, ax                 ; Return FALSE

DoneValidOffset:
        ret
ValidOffset     endp

;---------------------------------------------------------------------;
; This function tests a pointer first to see if the selector part is  ;
; valid.  And if so, it makes sure you can read at least cwSize bytes ;
; starting at *lp.                                                    ;
;                                                                     ;
;       BOOL ValidReadPtr(void FAR *lp, WORD cwSize);                 ;
;---------------------------------------------------------------------;
ValidReadPtr    proc    lp:Ptr, cwSize:Word
        verr    Word Ptr lp[2]         ; Is the selector valid?
        jnz     InvalidReadPtr         ; No, report this fact

        INVOKE  ValidOffset, lp, cwSize ; Report if offset is valid
        jmp     DoneValidReadPtr

InvalidReadPtr:
        xor     ax, ax                 ; Return FALSE

DoneValidReadPtr:
        ret
ValidReadPtr    endp


;---------------------------------------------------------------------;
; This function tests a pointer first to see if the selector part is  ;
; valid.  And if so, it makes sure you can write at least cwSize bytes ;
; starting at *lp.                                                    ;
;                                                                     ;
;       BOOL ValidWritePtr(void FAR *lp, WORD cwSize);                ;
;---------------------------------------------------------------------;
```

```
ValidWritePtr   proc    lp:Ptr, cwSize:Word
        verr    Word Ptr lp[2]          ; Is the selector valid?
        jnz     InvalidWritePtr         ; No, report this fact

        INVOKE  ValidOffset, lp, cwSize ; Report if offset is valid
        jmp     DoneValidWritePtr

InvalidWritePtr:
        xor     ax, ax                  ; Return FALSE

DoneValidWritePtr:
        ret
ValidWritePtr   endp

        end_
```

There are a few things in this program that may not be obvious. First, the Is286 function has code to check to see if the processor is an 80286 processor. You need to call this code to see whether or not 32-bit registers will be available. The actual code to test the processors is from an excellent book called *PC Magazine Programmer's Technical Reference: The Processor and Coprocessor*, by Robert L. Hummel, which is the best reference book we know of on the 80x86 processors.

The other thing that may be new to you is the INVOKE directive. This is a special directive added to MASM 6 that makes it very easy to call a high-level language function that expects its parameters to be passed on the stack. Since we wrote SegSize to use the stack for parameters, it was easiest to call it using the INVOKE directive. The INVOKE directive actually creates several assembly-language instructions that push the correct parameters onto the stack by using PUSH instructions, and then uses the CALL instruction to call your function. The advantage of using INVOKE over doing the PUSHes yourself is that the INVOKE directive will always produce the correct code based on the current language you have selected.

Summary

In this chapter we have given you just a glimpse of what you can do using assembly language in your Windows programs. As you have learned from the simple example in this chapter, you can write assembly-language subroutines quite simply for your C/C++ programs. But the interesting part is what you can do when your programs run in 386-Enhanced mode. In this mode you can write very fast assembly-language functions that treat large chunks of memory as a single segment. This is something that you cannot do directly from C/C++ with most of the current batch of compilers. If you need to perform an operation on a large block of data, the fastest code will be written in assembly language.

In this chapter we also provided functions you can use to test pointers to make sure they are valid. These functions will work with any version of Windows, or even in other protected-mode operating systems that run on the 80x86.

This is the last technical chapter in this book. In the next chapter you will find a bibliography of useful books where you can find more advanced information.

CHAPTER

Closing Words and Bibliography

By now you have seen many examples of assembly-language programs. Throughout this book we have constantly emphasized programming rather than the details of the 80x86 microprocessor inside your PC. As a result, you have not seen all of the 80x86 instructions, nor all the assembler directives. But most assembly language programs can be written with what you have learned here.

34

Topics Covered

80x86 Reference Books

DOS and ROM BIOS Programming

RAM Resident Programs

Advanced DOS Programming

Windows Programming

Software Design

Other References

Your best approach to learning more about writing assembly language programs is to take the programs in this book and modify them. If you think of a better way to write any part of Dskpatch, by all means try it. This is how we first learned to write programs. We found programs written in BASIC, and began to learn about the language itself by rewriting bits and pieces of those programs. You can do the same with Dskpatch.

After you have tried some of these examples, you will be ready to write your own programs. Don't start from scratch here, either; that is rather difficult for your first time out. To begin, use the programs in this book as a framework. Don't build a completely new structure or technique (your equivalent of modular design) until you feel comfortable with writing assembly language programs.

If you become enthralled by assembly language you will need a more complete book to use as a reference to the 80x86 instruction set. Following is a list of books we have read and liked that you may find useful as references. This list is by no means complete, as the books listed here are only ones that we have read. Also, some of the references are older than you might expect because we learned assembly language programming several years ago. Some of the books may now be out of print; check with your local booksellers.

80x86 Reference Books

The following three books are good programmer's references:

Hummel, Robert L., *PC Programmer's Technical Reference: The Processor and Coprocessor*, Ziff-Davis Press, 1992. This is the best book I have ever seen on the 80x86 processors. In fact, it is probably the only place where you will find all of the bugs in the various processors, as well as ways to work around the bugs. Highly recommended.

Microsoft's 80386/80486 Programming Guide, Microsoft Press, 1991. This book is not as good as Robert Hummel's book (see above), but it does a very nice job describing the protected mode.

iAPX 88 Book, Intel, 1981. This is the definitive sourcebook on the original 8088/8086, and a very good reference.

The following book is an introduction to the 8088 microprocessor, written by a member of the design team at Intel:

Morse, Stephen P., *The 8086/8088 Primer*, Hayden, 1982. As one of the designers at Intel, Morse provides many insights into the design of the 8088 and also talks about some of the design flaws and bugs in the 8088. While not very good as a reference, this book is complete and is readable and informative.

DOS and ROM BIOS Programming

The references in this section are useful to anyone programming the PC.

Norton, Peter and Richard Wilton, *The New Peter Norton Programmer's Guide to the IBM PC & PS/2*, Microsoft Press, 1988. Includes a complete reference to all DOS and BIOS functions, descriptions of important memory locations, and a host of other useful (or at least interesting) information.

Duncan, Ray, *Advanced MS-DOS*, Microsoft Press, 1986. This book covers almost everything you will want to know about using the DOS services in your programs. It also includes a number of sample programs. A nice companion to Peter's *Programmer's Guide.*

Brown, Ralf, and Jim Kyle, *PC Interrupts: A Programmer's Reference to BIOS, DOS, and Third-Party Calls*, Addison-Wesley, 1991. This is probably the best all-around reference to ROM-BIOS and DOS calls.

RAM Resident Programs

There aren't many good references for people who want to write RAM-resident programs because much of the material has not been published in a single place. But there are two good sources for information:

Duncan, Ray, Editor, *The MS-DOS Encyclopedia*, Microsoft Press, 1988. This book has a wealth of information. It covers many of the aspects of writing RAM-resident programs.

PC Magazine, published by Ziff Davis often publishes information on RAM-resident programs, as well as example programs. A subscription to this magazine will provide you with many good assembly-language programs.

Advanced DOS Programming

After you have started DOS programming, you will probably need to branch far beyond this book. The following books should help:

Duncan, Ray, Editor, *Extending DOS: A Programmer's Guide to Protected-Mode DOS*, Addison-Wesley, 1992. If you want to do protected-mode programming from DOS, you want this book. This explains all about EMS memory, extended memory, XMS, VCPI, DPMI, and so on.

Schulman, Andrew, et al, *Undocumented DOS: A Programmer's Guide to Reserved MS-DOS Functions and Data Structures*, Addison-Wesley, 1990. That about says it all for this book. Very interesting.

Windows Programming

There are a number of very good Windows programming books. Following are some of the best ones:

Petzold, Charles, *Programming Windows 3.1*, Microsoft Press, 1992. This is an excellent book that teaches Windows programming.

Norton, Peter and Paul Yao, *Peter Norton's Windows 3.1 Power Programming Techniques*, Bantam, 1992. This is another very good Windows programming book. You will find more advanced material in this book, as well as some good examples of using in-line assembly in Windows programs.

Richter, Jeffrey M., *Windows 3.1: A Developer's Guide*, M&T Books, 1992. This book is considered the standard reference book for more advanced material related to Windows programming.

Schulman, Andrew, et al, *Undocumented Windows*, Addison Wesley, 1992. This is a wonderful book that uncovers many of the previously undocumented parts of Windows.

Software Design

We have a few favorite books when it comes to software design. The books we recommend are a bit out of the ordinary, and well worth the read.

Brooks, Frederick P., Jr., *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, 1982. Everyone who is connected with a software project should read this book, especially your manager.

Normal, Donald A., *The Design of Everyday Things*, Basic Books, 1988. This book provides a lot of useful insight into what does and does not create problems with programs that interact with people.

Heckel, Paul, *The Elements of Friendly Software Design*, Sybex, 1991.

Other References

Startz, Richard, *8087 Applications and Programming for the IBM PC and Other PCs*, Brady Books, 1983 (Out of print). If you need to write programs that use the 8087 math coprocessor, this book should help.

Wilton, Richard, *Programmer's Guide to PC & PS/2 Video Systems*, Microsoft Press, 1987. This is a good book about writing programs for the EGA and VGA display adapters.

A P P E N D I X

Guide to the Disk

The companion disk to this book contains most of the Dskpatch examples you have seen in the preceding chapters, as well as an advanced version of the program that includes a lot of improvements. The files for Dskpatch are in two groups: the chapter examples in the CHAPS directory and the advanced Dskpatch program in the advanced directory. The file for the Disklite program discussed in Chapter 32 is included in the directory DISKLITE. The WINDOWS directory includes the file discussed in Chapter 33. Finally, a special group of assembly procedures written for C/C++ programming is included in the CLIB directory. This appendix will explain what's on the disk, and why.

A

Topics Covered

Chapter Examples

Advanced Version of Dskpatch

DISKLITE Program

Windows Code

C/C++ Libraries

Chapter Examples

All the chapter examples are from Chapters 9 through 27, 30, and 31, and they are in the CHAPS directory. The examples in earlier chapters are short enough so you can type them in quickly. But starting in Chapter 9, we began to build Dskpatch, which, by the end of this book had grown to nine different files. The files are all routines for Dskpatch except CLIB.ASM which is discussed in Chapter 31.

In any one chapter, only a few of these nine files changed. Since they do evolve throughout each chapter, however, there was not enough room on the disk to store each version of each example. So you will find the examples on the disk as they stand after each chapter. If we modify a program several times in, say, Chapter 19, the disk contains the final version.

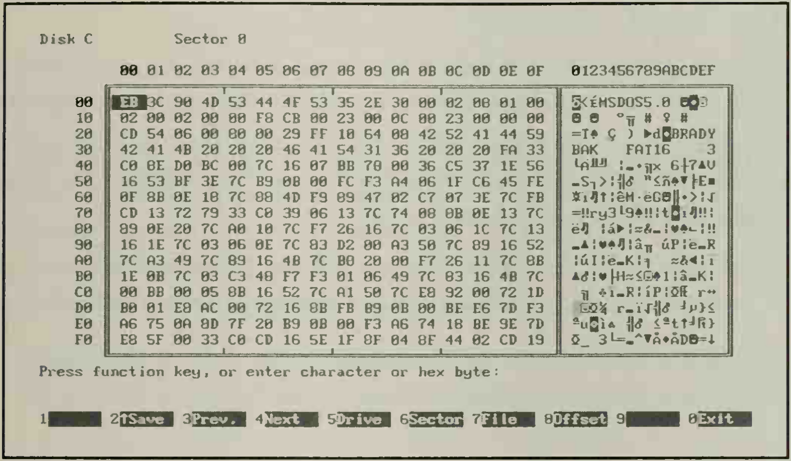
The table at the end of this appendix shows when each file changes. It also shows the name of the disk file for that chapter. If you want to make sure you are still on course, or you don't feel like typing in the changes for some chapter, just look at this table to find the names of the new files. Then you can either check your work or copy the file(s) to your disk.

The complete list of all the files on the companion disk that cover the chapter-by-chapter changes to Dskpatch is as follows:

VIDEO_9.ASM	VIDEO_16.ASM	DISP_S19.ASM	KBD_IO24.ASM
VIDEO_10.ASM	DISK_I16.ASM	KBD_IO19.ASM	DISPAT25.ASM
VIDEO_13.ASM	DSKPAT17.ASM	DISK_I19.ASM	DISPAT26.ASM
TEST13.ASM	DISP_S17.ASM	DISP_S21.ASM	DISK_I26.ASM
DISP_S14.ASM	CURSOR17.ASM	PHANTO21.ASM	PHANTO27.ASM
CURSOR14.ASM	VIDEO_17.ASM	VIDEO_21.ASM	DSKPAT30.ASM
VIDEO_14.ASM	DISK_I17.ASM	DISPAT22.ASM	KBD_IO30.ASM
DISP_S15.ASM	DISP_S18.ASM	EDITOR22.ASM	CURSOR30.ASM
DISK_I15.ASM	CURSOR18.ASM	PHANTO22.ASM	VIDEO_30.ASM
DISP_S16.ASM	VIDEO_18.ASM	KBD_IO23.ASM	CLIB.ASM
DSKPAT19.ASM	DISPAT19.ASM	TEST23.ASM	

Advanced Version of Dskpatch

As we said, the disk contains more than just the examples in this book. We did not really finish Dskpatch by the end of Chapter 27, and there are many things we should have put into Dskpatch to make it a usable program. The disk contains an almost-finished version in the ADVANCED directory as shown in Figure A-1.



```
Disk C          Sector 0

         00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F    0123456789ABCDEF

   00    EB 3C 90 4D 53 44 4F 53 35 2E 30 00 02 08 01 00    ═<éMSDOS5.0 ◘☻
   10    02 00 02 00 00 F8 CB 00 23 00 0C 00 23 00 00 00    ☻ ☻  °╥ # ♀ #
   20    CD 54 06 00 00 00 29 FF 10 64 00 42 52 41 44 59    =T♠ Ç ) ►d BRADY
   30    42 41 4B 20 20 20 46 41 54 31 36 20 20 20 FA 33    BAK   FAT16   ·3
   40    C0 8E D0 BC 00 7C 16 07 BB 78 00 36 C5 37 1E 56    └Ä╨╝ |▬•╗x 6┼7▲V
   50    16 53 BF 3E 7C B9 0B 00 FC F3 A4 06 1F C6 45 FE    ▬S┐>|╣♂ ⁿ≤ñ♠▼╞E■
   60    0F 8B 0E 18 7C 88 4D F9 89 47 02 C7 07 3E 7C FB    ☼ï♫↑|êM·ëG☻╟•>|√
   70    CD 13 72 79 33 C0 39 06 13 7C 74 08 0B 0E 13 7C    =‼ry3└9♠‼|t◘♂♫‼|
   80    89 0E 20 7C A0 10 7C F7 26 16 7C 03 06 1C 7C 13    ë♫ |á►|≈&▬|♥♠∟|‼
   90    16 1E 7C 03 06 0E 7C 83 D2 00 A3 50 7C 89 16 52    ▬▲|♥♠♫|â╥ úP|ë▬R
   A0    7C A3 49 7C 89 16 4B 7C B0 20 00 F7 26 11 7C 8B    |úI|ë▬K|░  ≈&◄|ï
   B0    1E 0B 7C 03 C3 48 F7 F3 01 06 49 7C 83 16 4B 7C    ▲♂|♥├H≈≤☺♠I|â▬K|
   C0    00 BB 00 05 8B 16 52 7C A1 50 7C E8 92 00 72 1D     ╗ ♣ï▬R|íP|Φ╞ r↔
   D0    B0 01 E8 AC 00 72 16 8B FB 89 0B 00 BE E6 7D F3    ░☺Φ¼ r▬ï√ë♂ ╛µ}≤
   E0    A6 75 0A 8D 7F 20 B9 0B 00 F3 A6 74 18 BE 9E 7D    ªu◙ì⌂ ╣♂ ≤ªt↑╛₧}
   F0    E8 5F 00 33 C0 CD 16 5E 1F 8F 04 8F 44 02 CD 19    Φ_ 3└=▬^▼Å♦ÅD☻=↓

Press function key, or enter character or hex byte:

1        2↑Save   3Prev.   4Next    5Drive   6Sector  7File    8Offset  9        0Exit
```

Figure A-1: The Advanced version of Dskpatch.

As it stands (in this book), Dskpatch can read only the next or previous sector. Thus, if you wanted to read sector 576, you would have to push the F4 key 575 times. And what if you wanted to look at sectors within a file? Right now, you would have to look at the directory sector and figure out where to look for the sectors of that file. The disk version of Dskpatch can read either absolute sectors, just as the book version can, or it can read sectors within a file. In its advanced form, Dskpatch is a very usable program.

The advanced version of Dskpatch has too many changes to describe in detail here, so we will take a quick look at the new functions we added to the disk version. You will find many of the changes by exploring Dskpatch and by making your own changes.

The advanced Dskpatch still has nine files, all of which you will find on the disk.

```
DSKPATCH.ASM  DISPATCH.ASM  DISP_SEC.ASM  KBD_IO.ASM
CURSOR.ASM    EDITOR.ASM    PHANTOM.ASM   VIDEO_IO.ASM
DISK_IO.ASM
```

MASM users can assemble the program with the included MAKEFILE and LINKINFO files; TASM users can assemble with the BORLAND. MAK file. You will also find an assembled and linked EXE version ready to run, so you can try out the new version without assembling it.

When you do, you will be able to tell that there are several improvements just by looking at the screen display. The advanced Dskpatch now uses eight function keys. That is more than you can remember if you don't use Dskpatch very often, so the advanced Dskpatch has a "key line" at the bottom of the display. A description of the function keys is as follows:

F2	Press the Shift key and F2 to write a sector back to the disk. We have covered this function in the book.
F3, F4	F3 reads the previous sector, and F4 reads the next sector. We have covered these two functions in the book.
F5	Changes the disk drive number or letter. Just press F5 and enter a letter, such as A (without a colon, :), or enter a drive number, such as 0. When you press the Enter key, Dskpatch will change drives and read a sector from the new disk drive. You may want to change Dskpatch so it does not read a new sector when you change drives. We have set it up so that it is very difficult to write a sector to the wrong disk.
F6	Changes the sector number. Just press F6 and type a sector number, in decimal. Dskpatch will read that sector.
F7	Changes Dskpatch to file mode. Just enter the file name and Dskpatch will read a sector from that file. From then on, F3 (Previous Sector) and F4 (Next

	Sector) read sectors from within that file. F5 ends file mode and switches back to absolute-sector mode.
F8	Asks for an offset within a file. This is just like F4 (Sector) except that it reads sectors within a file. If you enter an offset of 3, Dskpatch will read the fourth sector in your file.
F10	Exits from Dskpatch. If you accidentally press this key, you will find yourself back in DOS, and you will lose any changes that you have made to the last sector. You may want to change Dskpatch so that it asks if you really want to leave Dskpatch.

A number of other changes are not as obvious as those we just mentioned. For example, Dskpatch now scrolls the screen one line at a time. So, if you move the cursor to the bottom line of the display and press the Cursor-Down key, Dskpatch will scroll the display by one line, putting a new line at the bottom. In addition, some of the other keys on the keyboard also work now:

Home	Moves the phantom cursor to the top of the half-sector display and scrolls the display so you see the first half-sector.
End	Moves the phantom cursor to the bottom right of the half-sector display and scrolls the display so you see the second half-sector.
PgUp	Scrolls the half-sector display by four lines. This is a nice feature when you want to move partway through the sector display. If you press PgUp four times, you will see the last half sector.
PgDn	Scrolls the half-sector display by four lines in the opposite direction from PgUp.

You can modify the advanced Dskpatch to better suit your own needs. That is why the disk has all the source files for the advanced Dskpatch—so you can modify Dskpatch any way you like and learn from a complete example. For instance, you might spruce up the error-checking capabilities. As it stands, if pressing F4 causes you to fall off the end of a disk or file, Dskpatch does not reset the sector to the last sector on the disk or file. If you feel ambitious, see if you can modify Dskpatch so it catches and corrects such errors.

Or, you may want to speed up screen updates. To do this you would have to rewrite some of the procedures, such as WRITE_CHAR and WRITE_ATTRIBUTE_N_TIMES, to write directly to screen memory. Now, they use the very slow ROM BIOS routines. If you are really ambitious, try to write your own character-output routines that send characters to the screen quickly.

DISKLITE Program

The DISKLITE directory included just one file, DISKLITE.ASM. The file contains the complete program for building a RAM resident program that was developed in Chapter 31.

Windows Code

The WINDOWS directory contains two files. WINLIB.ASM contains the code from Chapter 33 that checks pointers to see if they are valid. It also reports the size of segments. WINTEST.C is a sample file used to demonstrate the function. Use MASM 6.

C/C++ Libraries

As mentioned in Chapter 31, you will also find the set of C/C++ libraries on the disk, which are also printed in Appendix C. These libraries provide mouse support, writing to your screen, and some keyboard input routines. They show you how you can write a set of assembly-language subroutines that work with any memory model. MASM 6 is needed to build the library file CLIB.LIB. A list of the files you will find in the CLIB directory is as follows:

```
CURSOR.ASM         FASTWRIT.ASM       HARDWARE.ASM       KBD_IO.ASM
MOUSE.ASM          MAKEFILE           SOCHALIB.INC
```

Number	DSKPATCH	DISPATCH	DISP_SEC	KBD_IO	CURSOR	EDITOR	PHANTOM	VIDEO_IO	DISK_IO	TEST
9								VIDEO_9 ASM		
10								VIDEO_10 ASM		
13								VIDEO_13 ASM		TEST13 ASM
14			DISP_S14 ASM		CURSOR14 ASM			VIDEO_14 ASM		
15			DISP_S15 ASM		→			→	DISK_I15 ASM	
16			DISP_S16 ASM		→			VIDEO_16 ASM	DISK_I16 ASM	
17	DSKPAT17 ASM		DISP_S17 ASM		CURSOR17 ASM			VIDEO_17 ASM	DISK_I17 ASM	
18	→		DISP_18 ASM		CURSOR18 ASM			VIDEO_18 ASM	→	
19	DSKPAT19 ASM	DISPAT19 ASM	DISP_19 ASM	KBD_IO19 ASM	→			→	DISK_I19 ASM	
21		→	DISP_S21 ASM	→	→		PHANTO21 ASM	VIDEO_21 ASM	→	
22		DISPAT22 ASM	→	→	→	EDITOR22 ASM	PHANTO22 ASM	→	→	
23			→	KBD_IO23 ASM	→	→	→	→	→	TEST23 ASM
24			→	KBD_IO24 ASM	→	→	→	→	→	
25		DISPAT25 ASM	→	→	→	→	→	→	→	
26		DISPAT26 ASM	→	→	→	→	→	→	DISK_I26 ASM	
27		→	→	→	→	→	PHANTO27 ASM	→		
30	DSKPAT30 ASM			KBD_IO30 ASM	CURSOR30 ASM			VIDEO_30 ASM		

A P P E N D I X

Listing of Dskpatch

This appendix contains the final version of Dskpatch. If you are writing your own programs, you will find many general-purpose procedures in this appendix that will help you on your way. We have included short descriptions of each procedure to help you find such procedures.

B

Topics Covered

Descriptions of Procedures

Dskpatch Make File

Dskpatch Linkinfo File

Program Listings for Dskpatch Procedures

Descriptions of Procedures

CURSOR.ASM

CLEAR_SCREEN Like the BASIC CLS command; clears the text screen.

CLEAR_TO_END_OF_LINE Clears all the characters from the cursor position to the end of the current line.

CURSOR_RIGHT Moves the cursor one character position to the right, without writing a space over the old character.

GOTO_XY Very much like the BASIC LOCATE command; moves the cursor on the screen.

SEND_CRLF Sends a carriage-return/line-feed pair of characters to the screen. This procedure simply moves the cursor to the start of the next line.

UPDATE_REAL_CURSOR Moves the real cursor to the location of the virtual cursor.

UPDATE_VIRTUAL_CURSOR Moves the virtual cursor to the position of the real cursor.

DISK_IO.ASM

NEXT_SECTOR Adds one to the current sector number, then reads that sector into memory and rewrites the Dskpatch screen.

PREVIOUS_SECTOR Reads the previous sector. That is, the procedure subtracts one from the old sector number (CURRENT_SECTOR_NO) and reads the new sector into the memory variable SECTOR. It also rewrites the screen display.

READ_SECTOR Reads one sector (512 bytes) from the disk into the memory buffer, SECTOR.

WRITE_SECTOR Writes one sector (512 bytes) from the memory buffer, SECTOR, to the disk.

DISPATCH.ASM

DISPATCHER The central dispatcher; reads characters from the keyboard and then calls on other procedures to do all the work of Dskpatch. Add any new commands to DISPATCH_TABLE in this file.

DISP_SEC.ASM

DISP_HALF_SECTOR Does the work of displaying all the hex and ASCII characters that appear in the half-sector display by calling DISP_LINE 16 times.

DISP_LINE Displays just 1 line of the half-sector display. DISP_HALF_SECTOR calls this procedure 16 times to display all 16 lines of the half-sector display.

INIT_SEC_DISP Initializes the half-sector display you see in Dskpatch. This procedure redraws the half-sector display, along with the boundaries and top hex numbers, but does not write the header or the editor prompt.

WRITE_HEADER Writes the header at the top of the screen you see in Dskpatch. There, the procedure displays the disk-drive number and the number of the sector you see in the half-sector display.

WRITE_PROMPT_LINE Writes a string at the prompt line, then clears the rest of the line to remove any characters from the old prompt.

WRITE_TOP_HEX_NUMBERS Writes the line of hex numbers across the top of the half-sector display. The procedure is not useful for much else.

DSKPATCH.ASM

DISK_PATCH The (very short) main program of Dskpatch. DISK_PATCH simply calls a number of other procedures, which do all the work. It also includes many of the definitions for variables that are used throughout Dskpatch.

EDITOR.ASM

EDIT_BYTE Edits a byte in the half-sector display by changing one byte both in memory (SECTOR) and on the screen. Dskpatch uses this procedure to change bytes in a sector.

WRITE_TO_MEMORY Called upon by EDIT_BYTE to change a single byte in SECTOR. This procedure changes the byte pointed to by the phantom cursor.

KBD_IO.ASM

BACK_SPACE Used by the READ_STRING procedure to delete one character, both from the screen and from the keyboard buffer, whenever you press the Backspace key.

CONVERT_HEX_DIGIT Converts a single ASCII character into its hexadecimal equivalent. For example, the procedure converts the letter A into the hex number 0AH. CONVERT_HEX_DIGIT works only with uppercase letters.

HEX_TO_BYTE Converts a two-character string of characters from a hexadecimal string, such as A5, into a single byte with that hex value. HEX_TO_BYTE expects the two characters to be digits or uppercase letters.

READ_BYTE Uses READ_STRING to read a string of characters. This procedure returns the special function key, a single character, or a hex byte if you typed a two-digit hex number.

READ_DECIMAL Reads an an unsigned decimal number from the keyboard, using READ_STRING to read the characters. READ_DECIMAL can read numbers from 0 to 65,535.

READ_KEY Reads a single key from the keyboard and returns 0 through 255 for ordinary characters, and 100h plus the scan code for special keys.

READ_STRING Reads a DOS-style string of characters from the keyboard. This procedure also reads special function keys, whereas the DOS READ_STRING function does not.

STRING_TO_UPPER A general-purpose procedure; converts a DOS-style string to all uppercase letters.

PHANTOM.ASM

ERASE_PHANTOM Removes the two phantom cursors from the screen by returning the character attribute to normal (7) for all characters under the phantom cursors.

MOV_TO_ASCII_POSITION Moves the real cursor to the start of the phantom cursor in the ASCII window of the half-sector display.

MOV_TO_HEX_POSITION Moves the real cursor to the start of the phantom cursor in the hex window of the half-sector display.

PHANTOM_DOWN Moves the phantom cursor down and scrolls the screen if you try to move past the 16th line of the half-sector display.

PHANTOM_LEFT Moves the phantom cursor left one entry, but not past the left side of the half-sector display.

PHANTOM_RIGHT Moves the phantom cursor right one entry, but not past the right side of the half-sector display.

PHANTOM_UP Moves the phantom cursor up one line in the half-sector display, or scrolls the display if you try to move the cursor off the top.

RESTORE_REAL_CURSOR Moves the cursor back to the position recorded by SAVE_REAL_CURSOR.

SAVE_REAL_CURSOR Saves the position of the real cursor in two variables. Call this procedure before you move the real cursor if you want to restore its position when you have finished making changes to the screen.

SCROLL_DOWN Rather than scrolling the half-sector display, displays the first half of the sector. You will find a more advanced version of SCROLL_DOWN on the disk available with this book. The advanced version scrolls the half-sector display by just one line.

SCROLL_UP Called by PHANTOM_DOWN when you try to move the phantom cursor off the bottom of the half-sector display. The version in this book doesn't actually scroll the screen; it writes the second half of the sector. On the disk, more advanced versions of SCROLL_UP and SCROLL_DOWN scroll the display by 1 line instead of 16.

WRITE_PHANTOM Draws the phantom cursors in the half-sector display: one in the hex window, and one in the ASCII window. This procedure simply changes the character attributes to 70H, to use black characters on a white background.

VIDEO_IO.ASM

Contains most of the general-purpose procedures you will want to use in your own programs.

INIT_WRITE_CHAR Call this procedure before you call any of the other procedures in this file. It initializes the data used by the routines that write directly to screen memory.

WRITE_ATTRIBUTE_N_TIMES A handy procedure you can use to change the attributes for a group of N characters. WRITE_PHANTOM uses this procedure to draw the phantom cursors, and ERASE_PHANTOM uses it to remove the phantom cursors.

WRITE_CHAR Writes a character to the screen. Since it uses the ROM BIOS routines, this procedure does not attach special meaning to any characters. So, a carriage-return character will appear on the screen as a musical note (the character for 0DH). Call SEND_CRLF if you want to move the cursor to the start of the next line.

WRITE_CHAR_N_TIMES Writes *N* copies of one character to the screen. This procedure is useful for drawing lines of characters, such as the ones used in patterns.

WRITE_DECIMAL Writes a word to the screen as an unsigned decimal number in the range 0 to 65,535.

WRITE_HEX Takes a one-byte number and writes it on the screen as a two-digit hex number.

WRITE_HEX_DIGIT Writes a single-digit hex number on the screen. This procedure converts a 4-bit nibble into the ASCII character and writes it to the screen.

WRITE_PATTERN Draws boxes around the half-sector display, as defined by a pattern. You can use WRITE_PATTERN to draw arbitrary patterns of characters on the screen.

WRITE_STRING A very useful, general-purpose procedure with which you can write a string of characters to the screen. The last character in your string must be a zero byte.

Dskpatch Make File

Following is the makefile that you can use to build Dskpatch automatically.

```
dskpatch.exe:   dskpatch.obj disk_io.obj disp_sec.obj video_io.obj cursor.obj \
                dispatch.obj kbd_io.obj phantom.obj editor.obj
        link @linkinfo

dskpatch.obj:   dskpatch.asm
        ml /c /Zi dskpatch.asm
```

```
disk_io.obj:     disk_io.asm
        ml /c /Zi disk_io.asm

disp_sec.obj:    disp_sec.asm
        ml /c /Zi disp_sec.asm

video_io.obj:    video_io.asm
        ml /c /Zi video_io.asm

cursor.obj:      cursor.asm
        ml /c /Zi cursor.asm

dispatch.obj:    dispatch.asm
        ml /c /Zi dispatch.asm

kbd_io.obj:      kbd_io.asm
        ml /c /Zi kbd_io.asm

phantom.obj:     phantom.asm
        ml /c /Zi phantom.asm

editor.obj:      editor.asm
        ml /c /Zi editor.asm
```

Dskpatch Linkinfo File

The linkinfo file is as follows:

```
DSKPATCH DISK_IO DISP_SEC VIDEO_IO CURSOR +
DISPATCH KBD_IO PHANTOM EDITOR
DSKPATCH
DSKPATCH /MAP;
```

Program Listings for Dskpatch Procedures

CURSOR.ASM

```
CR      EQU     13                      ;Carriage return
LF      EQU     10                      ;Line feed

.MODEL  SMALL
.CODE

        PUBLIC  CLEAR_SCREEN
;-----------------------------------------------------------------------;
; This procedure clears the entire screen.                              ;
;-----------------------------------------------------------------------;
CLEAR_SCREEN    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        XOR     AL,AL                   ;Blank entire window
        XOR     CX,CX                   ;Upper left corner is at (0,0)
        MOV     DH,24                   ;Bottom line of screen is line 24
        MOV     DL,79                   ;Right side is at column 79
        MOV     BH,7                    ;Use normal attribute for blanks
        MOV     AH,6                    ;Call for SCROLL-UP function
        INT     10h                     ;Clear the window
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CLEAR_SCREEN    ENDP


        PUBLIC  GOTO_XY
.DATA
        EXTRN   SCREEN_PTR:WORD ;Pointer to character under cursor
        EXTRN   SCREEN_X:BYTE, SCREEN_Y:BYTE
```

```
.CODE
;-----------------------------------------------------------------------;
; This procedure moves the cursor                                       ;
;                                                                       ;
; On entry:       DH      Row (Y)                                       ;
;                 DL      Column (X)                                    ;
;-----------------------------------------------------------------------;
GOTO_XY         PROC
        PUSH    AX
        PUSH    BX
        MOV     BH,0                    ;Display page 0
        MOV     AH,2                    ;Call for SET CURSOR POSITION
        INT     10h

        MOV     AL,DH                   ;Get the row number
        MOV     BL,80                   ;Multiply by 80 chars per line
        MUL     BL                      ;AX = row * 80
        ADD     AL,DL                   ;Add column
        ADC     AH,0                    ;AX = row * 80 + column
        SHL     AX,1                    ;Convert to a byte offset
        MOV     SCREEN_PTR,AX           ;Save the cursor offset
        MOV     SCREEN_X,DL             ;Save the cursor position
        MOV     SCREEN_Y,DH

        POP     BX
        POP     AX
        RET
GOTO_XY         ENDP


        PUBLIC  CURSOR_RIGHT
.DATA
        EXTRN   SCREEN_PTR:WORD ;Pointer to character under cursor
        EXTRN   SCREEN_X:BYTE, SCREEN_Y:BYTE
.CODE
;-----------------------------------------------------------------------;
; This procedure moves the cursor one position to the right or to the  ;
; next line if the cursor was at the end of a line.                     ;
;                                                                       ;
; Uses:           SEND_CRLF                                             ;
; Writes:         SCREEN_PTR, SCREEN_X, SCREEN_Y                        ;
;-----------------------------------------------------------------------;
CURSOR_RIGHT    PROC
        INC     SCREEN_PTR              ;Move to next character position (word)
        INC     SCREEN_PTR
        INC     SCREEN_X                ;Move to next column
        CMP     SCREEN_X,79             ;Make sure column <= 79
        JBE     OK
        CALL    SEND_CRLF               ;Go to next line
OK:
        RET
CURSOR_RIGHT    ENDP


        PUBLIC  UPDATE_REAL_CURSOR
;-----------------------------------------------------------------------;
; This procedure moves the real cursor to the current virtual cursor   ;
; position.  You'll want to call it just before you wait for keyboard  ;
; input.                                                                ;
;-----------------------------------------------------------------------;
UPDATE_REAL_CURSOR      PROC
        PUSH    DX
        MOV     DL,SCREEN_X             ;Get position of the virtual cursor
        MOV     DH,SCREEN_Y
        CALL    GOTO_XY                 ;Move real cursor to this position
        POP     DX
        RET
UPDATE_REAL_CURSOR      ENDP


        PUBLIC  UPDATE_VIRTUAL_CURSOR
;-----------------------------------------------------------------------;
; This procedure updates the position of our virtual cursor to agree   ;
; with the position of the real cursor.                                 ;
;-----------------------------------------------------------------------;
UPDATE_VIRTUAL_CURSOR   PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3            ;Ask for the cursor position
        XOR     BH,BH           ;On page 0
        INT     10h             ;Get cursor position into DH, DL
        CALL    GOTO_XY         ;Move virtual cursor to this position
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
UPDATE_VIRTUAL_CURSOR   ENDP
```

```
        PUBLIC  CLEAR_TO_END_OF_LINE
;-------------------------------------------------------------------------;
; This procedure clears the line from the current cursor position to     ;
; the end of that line.                                                   ;
;-------------------------------------------------------------------------;
CLEAR_TO_END_OF_LINE    PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     DL,SCREEN_X
        MOV     DH,SCREEN_Y
        MOV     AH,6                    ;Set up to clear to end of line
        XOR     AL,AL                   ;Clear window
        MOV     CH,DH                   ;All on same line
        MOV     CL,DL                   ;Start at the cursor position
        MOV     DL,79                   ;And stop at the end of the line
        MOV     BH,7                    ;Use normal attribute
        INT     10h
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
CLEAR_TO_END_OF_LINE    ENDP


        PUBLIC  SEND_CRLF
;-------------------------------------------------------------------------;
; This routine just sends a carriage return-line feed pair to the        ;
; display, using the DOS routines so that scrolling will be handled      ;
; correctly.                                                              ;
;                                                                         ;
; Uses: UPDATE_VIRTUAL_CURSOR                                             ;
;-------------------------------------------------------------------------;
SEND_CRLF       PROC
        PUSH    AX
        PUSH    DX
        MOV     AH,2
        MOV     DL,CR
        INT     21h
        MOV     DL,LF
        INT     21h
        CALL    UPDATE_VIRTUAL_CURSOR   ;Update position of virtual cursor
        POP     DX
        POP     AX
        RET
SEND_CRLF       ENDP


        END
```

DISK_IO.ASM

```
.MODEL  SMALL

.DATA

        EXTRN   SECTOR:BYTE
        EXTRN   DISK_DRIVE_NO:BYTE
        EXTRN   CURRENT_SECTOR_NO:WORD


.CODE

        PUBLIC  PREVIOUS_SECTOR
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC
.DATA
        EXTRN   CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE
.CODE
;-------------------------------------------------------------------;
; This procedure reads the previous sector, if possible.            ;
;                                                                   ;
; Uses:         WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP            ;
;               WRITE_PROMPT_LINE                                   ;
; Reads:        CURRENT_SECTOR_NO, EDITOR_PROMPT                    ;
; Writes:       CURRENT_SECTOR_NO                                   ;
;-------------------------------------------------------------------;
PREVIOUS_SECTOR         PROC
        PUSH    AX
        PUSH    DX
        MOV     AX,CURRENT_SECTOR_NO    ;Get current sector number
        OR      AX,AX                   ;Don't decrement if already 0
        JZ      DONT_DECREMENT_SECTOR
        DEC     AX
        MOV     CURRENT_SECTOR_NO,AX    ;Save new sector number
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP           ;Display new sector
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
DONT_DECREMENT_SECTOR:
        POP     DX
        POP     AX
        RET
PREVIOUS_SECTOR         ENDP

        PUBLIC  NEXT_SECTOR
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC
.DATA
        EXTRN   CURRENT_SECTOR_NO:WORD, EDITOR_PROMPT:BYTE
.CODE
;-------------------------------------------------------------------;
; Reads the next sector.                                            ;
;                                                                   ;
; Uses:         WRITE_HEADER, READ_SECTOR, INIT_SEC_DISP            ;
;               WRITE_PROMPT_LINE                                   ;
; Reads:        CURRENT_SECTOR_NO, EDITOR_PROMPT                    ;
; Writes:       CURRENT_SECTOR_NO                                   ;
;-------------------------------------------------------------------;
NEXT_SECTOR     PROC
        PUSH    AX
        PUSH    DX
        MOV     AX,CURRENT_SECTOR_NO
        INC     AX                      ;Move to next sector
        MOV     CURRENT_SECTOR_NO,AX
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP           ;Display new sector
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        POP     DX
        POP     AX
        RET
NEXT_SECTOR     ENDP

        PUBLIC  READ_SECTOR
;-------------------------------------------------------------------;
; This procedure reads one sector (512 bytes) into SECTOR.          ;
;                                                                   ;
; Reads:        CURRENT_SECTOR_NO, DISK_DRIVE_NO                    ;
; Writes:        SECTOR                                               ;
;---------------------------------------------------------------------;
READ_SECTOR      PROC
        PUSH     AX
        PUSH     BX
        PUSH     CX
        PUSH     DX
        MOV      AL,DISK_DRIVE_NO           ;Drive number
        MOV      CX,1                       ;Read only 1 sector
        MOV      DX,CURRENT_SECTOR_NO       ;Logical sector number
        LEA      BX,SECTOR                  ;Where to store this sector
        INT      25h                        ;Read the sector
        POPF                                ;Discard flags put on stack by DOS
        POP      DX
        POP      CX
        POP      BX
        POP      AX
        RET
READ_SECTOR      ENDP


        PUBLIC   WRITE_SECTOR
;---------------------------------------------------------------------;
; This procedure writes the sector back to the disk.                  ;
;                                                                     ;
; Reads:         DISK_DRIVE_NO, CURRENT_SECTOR_NO, SECTOR             ;
;---------------------------------------------------------------------;
WRITE_SECTOR     PROC
        PUSH     AX
        PUSH     BX
        PUSH     CX
        PUSH     DX
        MOV      AL,DISK_DRIVE_NO           ;Drive number
        MOV      CX,1                       ;Write 1 sector
        MOV      DX,CURRENT_SECTOR_NO       ;Logical sector
        LEA      BX,SECTOR
        INT      26h                        ;Write the sector to disk
        POPF                                ;Discard the flag information
        POP      DX
        POP      CX
        POP      BX
        POP      AX
        RET
WRITE_SECTOR     ENDP


        END
```

DISPATCH.ASM

```
.MODEL  SMALL

.CODE
        EXTRN   NEXT_SECTOR:PROC                        ;In DISK_IO.ASM
        EXTRN   PREVIOUS_SECTOR:PROC                    ;In DISK_IO.ASM
        EXTRN   PHANTOM_UP:PROC, PHANTOM_DOWN:PROC      ;In PHANTOM.ASM
        EXTRN   PHANTOM_LEFT:PROC, PHANTOM_RIGHT:PROC
        EXTRN   WRITE_SECTOR:PROC                       ;In DISK_IO.ASM
.DATA
;----------------------------------------------------------------------;
; This table contains the legal extended ASCII keys and the addresses  ;
; of the procedures that should be called when each key is pressed.    ;
;                                                                      ;
; The format of the table is                                           ;
;               DB      72              ;Extended code for cursor up   ;
;               DW      OFFSET _TEXT:PHANTOM_UP                        ;
;----------------------------------------------------------------------;
DISPATCH_TABLE  LABEL   BYTE
        DB      61                              ;F3
        DW      OFFSET _TEXT:PREVIOUS_SECTOR
        DB      62                              ;F4
        DW      OFFSET _TEXT:NEXT_SECTOR
        DB      72                              ;Cursor up
        DW      OFFSET _TEXT:PHANTOM_UP
        DB      80                              ;Cursor down
        DW      OFFSET _TEXT:PHANTOM_DOWN
        DB      75                              ;Cursor left
        DW      OFFSET _TEXT:PHANTOM_LEFT
        DB      77                              ;Cursor right
        DW      OFFSET _TEXT:PHANTOM_RIGHT
        DW      85                              ;Shift F2
        DW      OFFSET _TEXT:WRITE_SECTOR
        DB      0                               ;End of the table

.CODE

        PUBLIC  DISPATCHER
        EXTRN   READ_BYTE:PROC, EDIT_BYTE:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC
.DATA
        EXTRN   EDITOR_PROMPT:BYTE
```

```
.CODE
;-------------------------------------------------------------------;
; This is the central dispatcher. During normal editing and viewing,   ;
; this procedure reads characters from the keyboard and, if the char   ;
; is a command key (such as a cursor key), DISPATCHER calls the        ;
; procedures that do the actual work.  This dispatching is done for    ;
; special keys listed in the table DISPATCH_TABLE, where the procedure ;
; addresses are stored just after the key names.                       ;
;                                                                      ;
; If the character is not a special key, then it should be placed      ;
; directly into the sector buffer--this is the editing mode.           ;
;                                                                      ;
; Uses:         READ_BYTE, EDIT_BYTE, WRITE_PROMPT_LINE                ;
; Reads:        EDITOR_PROMPT                                          ;
;-------------------------------------------------------------------;
DISPATCHER      PROC
        PUSH    AX
        PUSH    BX
        PUSH    DX
DISPATCH_LOOP:
        CALL    READ_BYTE               ;Read character into AX
        OR      AH,AH                   ;AX = -1 if no character read, 1
                                        ; for an extended code.
        JS      NO_CHARS_READ           ;No character read, try again
        JNZ     SPECIAL_KEY             ;Read extended code
        MOV     DL,AL
        CALL    EDIT_BYTE               ;Was normal character, edit byte
        JMP     DISPATCH_LOOP           ;Read another character

SPECIAL_KEY:
        CMP     AL,68                   ;F10--exit?
        JE      END_DISPATCH            ;Yes, leave
                                        ;Use BX to look through table
        LEA     BX,DISPATCH_TABLE
SPECIAL_LOOP:
        CMP     BYTE PTR [BX],0         ;End of table?
        JE      NOT_IN_TABLE            ;Yes, key was not in the table
        CMP     AL,[BX]                 ;Is it this table entry?
        JE      DISPATCH                ;Yes, then dispatch
        ADD     BX,3                    ;No, try next entry
        JMP     SPECIAL_LOOP            ;Check next table entry

DISPATCH:
        INC     BX                      ;Point to address of procedure
        CALL    WORD PTR [BX]           ;Call procedure
        JMP     DISPATCH_LOOP           ;Wait for another key
```

```
NOT_IN_TABLE:                           ;Do nothing, just read next character
        JMP     DISPATCH_LOOP

NO_CHARS_READ:
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE       ;Erase any invalid characters typed
        JMP     DISPATCH_LOOP           ;Try again

END_DISPATCH:
        POP     DX
        POP     BX
        POP     AX
        RET
DISPATCHER      ENDP


        END
```

DISP_SEC.ASM

```
.MODEL  SMALL


;-----------------------------------------------------------------------;
; Graphics characters for border of sector.                             ;
;-----------------------------------------------------------------------;
VERTICAL_BAR    EQU     0BAh
HORIZONTAL_BAR  EQU     0CDh
UPPER_LEFT      EQU     0C9h
UPPER_RIGHT     EQU     0BBh
LOWER_LEFT      EQU     0C8h
LOWER_RIGHT     EQU     0BCh
TOP_T_BAR       EQU     0CBh
BOTTOM_T_BAR    EQU     0CAh
TOP_TICK        EQU     0D1h
BOTTOM_TICK     EQU     0CFh


.DATA

TOP_LINE_PATTERN        LABEL   BYTE
        DB      ' ',7
        DB      UPPER_LEFT,1
        DB      HORIZONTAL_BAR,12
        DB      TOP_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      TOP_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      TOP_TICK,1
        DB      HORIZONTAL_BAR,12
        DB      TOP_T_BAR,1
        DB      HORIZONTAL_BAR,18
        DB      UPPER_RIGHT,1
        DB      0
BOTTOM_LINE_PATTERN     LABEL   BYTE
        DB      ' ',7
        DB      LOWER_LEFT,1
        DB      HORIZONTAL_BAR,12
        DB      BOTTOM_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      BOTTOM_TICK,1
        DB      HORIZONTAL_BAR,11
        DB      BOTTOM_TICK,1
        DB      HORIZONTAL_BAR,12
        DB      BOTTOM_T_BAR,1
        DB      HORIZONTAL_BAR,18
        DB      LOWER_RIGHT,1
        DB      0


.DATA?
        EXTRN   SECTOR:BYTE

.CODE

        PUBLIC  INIT_SEC_DISP
        EXTRN   WRITE_PATTERN:PROC, SEND_CRLF:PROC
        EXTRN   GOTO_XY:PROC, WRITE_PHANTOM:PROC
.DATA
        EXTRN   LINES_BEFORE_SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
.CODE
;-----------------------------------------------------------------------;
; This procedure initializes the half-sector display.                   ;
;                                                                       ;
; Uses:         WRITE_PATTERN, SEND_CRLF, DISP_HALF_SECTOR              ;
;               WRITE_TOP_HEX_NUMBERS, GOTO_XY, WRITE_PHANTOM           ;
; Reads:          TOP_LINE_PATTERN, BOTTOM_LINE_PATTERN                    ;
;                 LINES_BEFORE_SECTOR                                      ;
; Writes:         SECTOR_OFFSET                                            ;
;----------------------------------------------------------------------;
INIT_SEC_DISP     PROC
        PUSH      DX
        XOR       DL,DL                        ;Move cursor into position
        MOV       DH,LINES_BEFORE_SECTOR
        CALL      GOTO_XY
        CALL      WRITE_TOP_HEX_NUMBERS
        LEA       DX,TOP_LINE_PATTERN
        CALL      WRITE_PATTERN
        CALL      SEND_CRLF
        XOR       DX,DX                        ;Start at the beginning of the sector
        MOV       SECTOR_OFFSET,DX             ;Set sector offset to 0
        CALL      DISP_HALF_SECTOR
        LEA       DX,BOTTOM_LINE_PATTERN
        CALL      WRITE_PATTERN
        CALL      WRITE_PHANTOM                ;Display the phantom cursor
        POP       DX
        RET
INIT_SEC_DISP     ENDP


        PUBLIC    WRITE_HEADER
.DATA
        EXTRN     HEADER_LINE_NO:BYTE
        EXTRN     HEADER_PART_1:BYTE
        EXTRN     HEADER_PART_2:BYTE
        EXTRN     DISK_DRIVE_NO:BYTE
        EXTRN     CURRENT_SECTOR_NO:WORD
.CODE
        EXTRN     WRITE_STRING:PROC, WRITE_DECIMAL:PROC
        EXTRN     GOTO_XY:PROC, CLEAR_TO_END_OF_LINE:PROC
;----------------------------------------------------------------------;
; This procedure writes the header with disk-drive and sector number.  ;
;                                                                      ;
; Uses:           GOTO_XY, WRITE_STRING, WRITE_CHAR, WRITE_DECIMAL     ;
;                 CLEAR_TO_END_OF_LINE                                 ;
; Reads:          HEADER_LINE_NO, HEADER_PART_1, HEADER_PART_2         ;
;                 DISK_DRIVE_NO, CURRENT_SECTOR_NO                     ;
;----------------------------------------------------------------------;
WRITE_HEADER      PROC
        PUSH      DX
        XOR       DL,DL                        ;Move cursor to header line number
        MOV       DH,HEADER_LINE_NO
        CALL    GOTO_XY
        LEA     DX,HEADER_PART_1
        CALL    WRITE_STRING
        MOV     DL,DISK_DRIVE_NO
        ADD     DL,'A'                  ;Print drives A, B, ...
        CALL    WRITE_CHAR
        LEA     DX,HEADER_PART_2
        CALL    WRITE_STRING
        MOV     DX,CURRENT_SECTOR_NO
        CALL    WRITE_DECIMAL
        CALL    CLEAR_TO_END_OF_LINE    ;Clear rest of sector number
        POP     DX
        RET
WRITE_HEADER    ENDP


        EXTRN   WRITE_CHAR_N_TIMES:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC
        EXTRN   WRITE_HEX_DIGIT:PROC, SEND_CRLF:PROC
;-------------------------------------------------------------------;
; This procedure writes the index numbers (0 through F) at the top of ;
; the half-sector display.                                          ;
;                                                                   ;
; Uses:         WRITE_CHAR_N_TIMES, WRITE_HEX, WRITE_CHAR           ;
;               WRITE_HEX_DIGIT, SEND_CRLF                          ;
;-------------------------------------------------------------------;
WRITE_TOP_HEX_NUMBERS   PROC
        PUSH    CX
        PUSH    DX
        MOV     DL,' '                  ;Write 9 spaces for left side
        MOV     CX,9
        CALL    WRITE_CHAR_N_TIMES
        XOR     DH,DH                   ;Start with 0
HEX_NUMBER_LOOP:
        MOV     DL,DH
        CALL    WRITE_HEX
        MOV     DL,' '
        CALL    WRITE_CHAR
        INC     DH
        CMP     DH,10h                  ;Done yet?
        JB      HEX_NUMBER_LOOP

        MOV     DL,' '                  ;Write hex numbers over ASCII window
        MOV     CX,2
        CALL    WRITE_CHAR_N_TIMES
        XOR     DL,DL
```

```
HEX_DIGIT_LOOP:
        CALL    WRITE_HEX_DIGIT
        INC     DL
        CMP     DL,10h
        JB      HEX_DIGIT_LOOP
        CALL    SEND_CRLF
        POP     DX
        POP     CX
        RET
WRITE_TOP_HEX_NUMBERS   ENDP


        PUBLIC  DISP_HALF_SECTOR
        EXTRN   SEND_CRLF:PROC
;-----------------------------------------------------------------------;
;  This procedure displays half a sector (256 bytes)                   ;
;                                                                       ;
; On entry:     DS:DX   Offset into sector, in bytes -- should be       ;
;                       multiple of 16.                                 ;
;                                                                       ;
; Uses:         DISP_LINE, SEND_CRLF                                    ;
;-----------------------------------------------------------------------;
DISP_HALF_SECTOR        PROC
        PUSH    CX
        PUSH    DX
        MOV     CX,16                   ;Display 16 lines
HALF_SECTOR:
        CALL    DISP_LINE
        CALL    SEND_CRLF
        ADD     DX,16
        LOOP    HALF_SECTOR
        POP     DX
        POP     CX
        RET
DISP_HALF_SECTOR        ENDP


        PUBLIC  DISP_LINE
        EXTRN   WRITE_HEX:PROC
        EXTRN   WRITE_CHAR:PROC
        EXTRN   WRITE_CHAR_N_TIMES:PROC
;-----------------------------------------------------------------------;
; This procedure displays one line of data, or 16 bytes, first in hex,  ;
; then in ASCII.                                                        ;
;                                                                       ;
; On entry:     DS:DX   Offset into sector, in bytes.                 ;
;                                                                     ;
; Uses:         WRITE_CHAR, WRITE_HEX, WRITE_CHAR_N_TIMES             ;
; Reads:        SECTOR                                                ;
;---------------------------------------------------------------------;
DISP_LINE       PROC
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     BX,DX                   ;Offset is more useful in BX
        MOV     DL,' '
        MOV     CX,3                    ;Write 3 spaces before line
        CALL    WRITE_CHAR_N_TIMES
                                        ;Write offset in hex
        CMP     BX,100h                 ;Is the first digit a 1?
        JB      WRITE_ONE               ;No, white space already in DL
        MOV     DL,'1'                  ;Yes, then place '1' into DL for output
WRITE_ONE:
        CALL    WRITE_CHAR
        MOV     DL,BL                   ;Copy lower byte into DL for hex output
        CALL    WRITE_HEX
                                        ;Write separator
        MOV     DL,' '
        CALL    WRITE_CHAR
        MOV     DL,VERTICAL_BAR         ;Draw left side of box
        CALL    WRITE_CHAR
        MOV     DL,' '
        CALL    WRITE_CHAR
                                        ;Now write out 16 bytes
        MOV     CX,16                   ;Dump 16 bytes
        PUSH    BX                      ;Save the offset for ASCII_LOOP
HEX_LOOP:
        MOV     DL,SECTOR[BX]           ;Get 1 byte
        CALL    WRITE_HEX               ;Dump this byte in hex
        MOV     DL,' '                  ;Write a space between numbers
        CALL    WRITE_CHAR
        INC     BX
        LOOP    HEX_LOOP

        MOV     DL,VERTICAL_BAR         ;Write separator
        CALL    WRITE_CHAR
        MOV     DL,' '                  ;Add another space before characters
        CALL    WRITE_CHAR
        MOV     CX,16
        POP     BX                      ;Get back offset into SECTOR
```

```
ASCII_LOOP:
        MOV     DL,SECTOR[BX]
        CALL    WRITE_CHAR
        INC     BX
        LOOP    ASCII_LOOP

        MOV     DL,' '                  ;Draw right side of box
        CALL    WRITE_CHAR
        MOV     DL,VERTICAL_BAR
        CALL    WRITE_CHAR

        POP     DX
        POP     CX
        POP     BX
        RET
DISP_LINE       ENDP


        PUBLIC  WRITE_PROMPT_LINE
        EXTRN   CLEAR_TO_END_OF_LINE:PROC, WRITE_STRING:PROC
        EXTRN   GOTO_XY:PROC
.DATA
        EXTRN   PROMPT_LINE_NO:BYTE
.CODE
;-------------------------------------------------------------------------;
; This procedure writes the prompt line to the screen and clears the     ;
; end of the line.                                                        ;
;                                                                         ;
; On entry:     DS:DX   Address of the prompt-line message                ;
;                                                                         ;
; Uses:         WRITE_STRING, CLEAR_TO_END_OF_LINE, GOTO_XY               ;
; Reads:        PROMPT_LINE_NO                                            ;
;-------------------------------------------------------------------------;
WRITE_PROMPT_LINE       PROC
        PUSH    DX
        XOR     DL,DL                   ;Write the prompt line and
        MOV     DH,PROMPT_LINE_NO       ; move the cursor there
        CALL    GOTO_XY
        POP     DX
        CALL    WRITE_STRING
        CALL    CLEAR_TO_END_OF_LINE
        RET
WRITE_PROMPT_LINE       ENDP


        END
```

DSKPATCH.ASM

```
DOSSEG
.MODEL  SMALL

.STACK

.DATA

        PUBLIC  SECTOR_OFFSET
;-------------------------------------------;
; SECTOR_OFFSET is the offset of the half-     ;
; sector display into the full sector.  It must;
; be a multiple of 16, and not greater than 256;
;-------------------------------------------;
SECTOR_OFFSET   DW      0

        PUBLIC  CURRENT_SECTOR_NO, DISK_DRIVE_NO
CURRENT_SECTOR_NO       DW      0               ;Initially sector 0
DISK_DRIVE_NO           DB      0               ;Initially Drive A:

        PUBLIC  LINES_BEFORE_SECTOR, HEADER_LINE_NO
        PUBLIC  HEADER_PART_1, HEADER_PART_2
;-------------------------------------------;
; LINES_BEFORE_SECTOR is the number of lines   ;
; at the top of the screen before the half-    ;
; sector display.                              ;
;-------------------------------------------;
LINES_BEFORE_SECTOR     DB      2
HEADER_LINE_NO          DB      0
HEADER_PART_1           DB      'Disk ',0
HEADER_PART_2           DB      '         Sector ',0
        PUBLIC  PROMPT_LINE_NO, EDITOR_PROMPT
PROMPT_LINE_NO          DB      21
EDITOR_PROMPT           DB      'Press function key, or enter'
                        DB      ' character or hex byte: ',0

.DATA?

        PUBLIC  SECTOR
;-------------------------------------------;
; The entire sector (up to 8192 bytes) is      ;
; stored in this part of memory.               ;
;-------------------------------------------;
SECTOR  DB      8192 DUP (?)

.CODE
```

```
        EXTRN   CLEAR_SCREEN:PROC, READ_SECTOR:PROC
        EXTRN   INIT_SEC_DISP:PROC, WRITE_HEADER:PROC
        EXTRN   WRITE_PROMPT_LINE:PROC, DISPATCHER:PROC
        EXTRN   INIT_WRITE_CHAR:PROC
DISK_PATCH      PROC
        MOV     AX,DGROUP               ;Put data segment into AX
        MOV     DS,AX                   ;Set DS to point to data

        CALL    INIT_WRITE_CHAR
        CALL    CLEAR_SCREEN
        CALL    WRITE_HEADER
        CALL    READ_SECTOR
        CALL    INIT_SEC_DISP
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        CALL    DISPATCHER

        MOV     AH,4Ch                  ;Return to DOS
        INT     21h
DISK_PATCH      ENDP


        END     DISK_PATCH
```

EDITOR.ASM

```
.MODEL  SMALL

.CODE

.DATA
        EXTRN   SECTOR:BYTE
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   PHANTOM_CURSOR_X:BYTE
        EXTRN   PHANTOM_CURSOR_Y:BYTE
.CODE
;--------------------------------------------------------------------;
; This procedure writes one byte to SECTOR, at the memory location   ;
; pointed to by the phantom cursor.                                  ;
;                                                                    ;
; On entry:     DL      Byte to write to SECTOR                      ;
;                                                                    ;
; The offset is calculated by                                        ;
;   OFFSET = SECTOR_OFFSET + (16 * PHANTOM_CURSOR_Y) + PHANTOM_CURSOR_X;
;                                                                    ;
; Reads:         PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y, SECTOR_OFFSET      ;
; Writes:        SECTOR                                              ;
;--------------------------------------------------------------------;
WRITE_TO_MEMORY PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,SECTOR_OFFSET
        MOV     AL,PHANTOM_CURSOR_Y
        XOR     AH,AH
        MOV     CL,4                    ;Multiply PHANTOM_CURSOR_Y by 16
        SHL     AX,CL
        ADD     BX,AX                   ;BX = SECTOR_OFFSET + (16 * Y)
        MOV     AL,PHANTOM_CURSOR_X
        XOR     AH,AH
        ADD     BX,AX                   ;That's the address!
        MOV     SECTOR[BX],DL           ;Now, store the byte
        POP     CX
        POP     BX
        POP     AX
        RET
WRITE_TO_MEMORY ENDP


        PUBLIC  EDIT_BYTE
        EXTRN   SAVE_REAL_CURSOR:PROC, RESTORE_REAL_CURSOR:PROC
        EXTRN   MOV_TO_HEX_POSITION:PROC, MOV_TO_ASCII_POSITION:PROC
        EXTRN   WRITE_PHANTOM:PROC, WRITE_PROMPT_LINE:PROC
        EXTRN   CURSOR_RIGHT:PROC, WRITE_HEX:PROC, WRITE_CHAR:PROC
.DATA
        EXTRN   EDITOR_PROMPT:BYTE
.CODE
;--------------------------------------------------------------------;
; This procedure changes a byte in memory and on the screen.          ;
;                                                                    ;
; On entry:      DL      Byte to write into SECTOR, and change on screen;
;                                                                    ;
; Uses:          SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR                ;
;                MOV_TO_HEX_POSITION, MOV_TO_ASCII_POSITION           ;
;                WRITE_PHANTOM, WRITE_PROMPT_LINE, CURSOR_RIGHT       ;
;                WRITE_HEX, WRITE_CHAR, WRITE_TO_MEMORY               ;
; Reads:         EDITOR_PROMPT                                       ;
;--------------------------------------------------------------------;
```

```
EDIT_BYTE       PROC
        PUSH    DX
        CALL    SAVE_REAL_CURSOR
        CALL    MOV_TO_HEX_POSITION     ;Move to the hex number in the
        CALL    CURSOR_RIGHT            ; hex window
        CALL    WRITE_HEX               ;Write the new number
        CALL    MOV_TO_ASCII_POSITION   ;Move to the char. in the ASCII window
        CALL    WRITE_CHAR              ;Write the new character
        CALL    RESTORE_REAL_CURSOR     ;Move cursor back where it belongs
        CALL    WRITE_PHANTOM           ;Rewrite the phantom cursor
        CALL    WRITE_TO_MEMORY         ;Save this new byte in SECTOR
        LEA     DX,EDITOR_PROMPT
        CALL    WRITE_PROMPT_LINE
        POP     DX
        RET
EDIT_BYTE       ENDP


        END
```

KBD_IO.ASM

```
.MODEL  SMALL

BS      EQU     8                       ;Backspace character
CR      EQU     13                      ;Carriage-return character
ESCAPE  EQU     27                      ;Escape character

.DATA

KEYBOARD_INPUT  LABEL   BYTE
CHAR_NUM_LIMIT  DB      0               ;Length of input buffer
NUM_CHARS_READ  DB      0               ;Number of characters read
CHARS           DB      80 DUP (0)      ;A buffer for keyboard input

.CODE

        PUBLIC  STRING_TO_UPPER
;-------------------------------------------------------------------;
; This procedure converts the string, using the DOS format for strings,;
; to all uppercase letters.                                         ;
;                                                                   ;
; On entry:     DS:DX   Address of string buffer                    ;
;-------------------------------------------------------------------;
```

```
STRING_TO_UPPER PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        MOV     BX,DX
        INC     BX                      ;Point to character count
        MOV     CL,[BX]                 ;Charac. count in 2nd byte of buffer
        XOR     CH,CH                   ;Clear upper byte of count
UPPER_LOOP:
        INC     BX                      ;Point to next character in buffer
        MOV     AL,[BX]
        CMP     AL,'a'                  ;See if it is a lowercase letter
        JB      NOT_LOWER               ;Nope
        CMP     AL,'z'
        JA      NOT_LOWER
        ADD     AL,'A'-'a'              ;Convert to uppercase letter
        MOV     [BX],AL
NOT_LOWER:
        LOOP    UPPER_LOOP
        POP     CX
        POP     BX
        POP     AX
        RET
STRING_TO_UPPER ENDP

;-----------------------------------------------------------------------;
; This procedure converts a character from ASCII (hex) to a nibble     ;
; (4 bits).                                                             ;
;                                                                       ;
; On entry:     AL      Character to convert                            ;
; Returns:      AL      Nibble                                          ;
;               CF      Set for error, cleared otherwise                ;
;-----------------------------------------------------------------------;
CONVERT_HEX_DIGIT       PROC
        CMP     AL,'0'                  ;Is it a legal digit?
        JB      BAD_DIGIT               ;Nope
        CMP     AL,'9'                  ;Not sure yet
        JA      TRY_HEX                 ;Might be hex digit
        SUB     AL,'0'                  ;Is decimal digit, convert to nibble
        CLC                             ;Clear the carry, no error
        RET
```

```
TRY_HEX:
        CMP     AL,'A'                  ;Not sure yet
        JB      BAD_DIGIT               ;Not hex
        CMP     AL,'F'                  ;Not sure yet
        JA      BAD_DIGIT               ;Not hex
        SUB     AL,'A'-10               ;Is hex, convert to nibble
        CLC                             ;Clear the carry, no error
        RET
BAD_DIGIT:
        STC                             ;Set the carry, error
        RET
CONVERT_HEX_DIGIT       ENDP

        PUBLIC  HEX_TO_BYTE
;-------------------------------------------------------------------;
; This procedure converts the two characters at DS:DX from hex to one ;
; byte.                                                             ;
;                                                                   ;
; On entry:     DS:DX   Address of two characters for hex number    ;
; Returns:      AL      Byte                                        ;
;               CF      Set for error, clear if no error            ;
;                                                                   ;
; Uses:         CONVERT_HEX_DIGIT                                   ;
;-------------------------------------------------------------------;
HEX_TO_BYTE     PROC
        PUSH    BX
        PUSH    CX
        MOV     BX,DX                   ;Put address in BX for indirect addr
        MOV     AL,[BX]                 ;Get first digit
        CALL    CONVERT_HEX_DIGIT
        JC      BAD_HEX                 ;Bad hex digit if carry set
        MOV     CX,4                    ;Now multiply by 16
        SHL     AL,CL
        MOV     AH,AL                   ;Retain a copy
        INC     BX                      ;Get second digit
        MOV     AL,[BX]
        CALL    CONVERT_HEX_DIGIT
        JC      BAD_HEX                 ;Bad hex digit if carry set
        OR      AL,AH                   ;Combine two nibbles
        CLC                             ;Clear carry for no error
DONE_HEX:
        POP     CX
        POP     BX
        RET
```

```
BAD_HEX:
        STC                             ;Set carry for error
        JMP     DONE_HEX
HEX_TO_BYTE     ENDP


        PUBLIC  READ_STRING
        EXTRN   WRITE_CHAR:PROC
        EXTRN   UPDATE_REAL_CURSOR:PROC
;-------------------------------------------------------------------------;
; This procedure performs a function very similar to the DOS 0Ah          ;
; function.  But this function will return a special character if a      ;
; function or keyboard key is pressed--no return for these keys.  And     ;
; ESCAPE will erase the input and start over again.                       ;
;                                                                         ;
;       DS:DX   Address for keyboard buffer.  The first byte must         ;
;               contain the maximum number of characters to read (plus    ;
;               one for the return).  And the second byte will be used    ;
;               by this procedure to return the number of characters      ;
;               actually read.                                            ;
;                       0       No characters read                        ;
;                       -1      One special character read                ;
;                       otherwise number actually read (not including     ;
;                               Enter key)                                ;
;                                                                         ;
; Uses: BACK_SPACE, WRITE_CHAR, UPDATE_REAL_CURSOR                        ;
;-------------------------------------------------------------------------;
READ_STRING     PROC    PROC
        PUSH    AX
        PUSH    BX
        PUSH    SI
        MOV     SI,DX                   ;Use SI for index register and

START_OVER:
        MOV     BX,2                    ;BX for offset to beginning of buffer

READ_LOOP:
        CALL    UPDATE_REAL_CURSOR      ;Move to position of virtual cursor
        CALL    READ_KEY                ;Read one key from the keyboard
        OR      AH,AH                   ;Is character extended ASCII?
        JNZ     EXTENDED                ;Yes, then process it.
STRING_NOT_EXTENDED:                    ;No, see what char it is
        CMP     AL,CR                   ;Is this a carriage return?
        JE      END_INPUT               ;Yes, we are done with input
        CMP     AL,BS                   ;Is it a backspace character?
        JNE     NOT_BS                  ;Nope
        CALL    BACK_SPACE              ;Yes, delete character
        JMP     READ_LOOP               ;Read the next character

NOT_BS: CMP     AL,ESCAPE               ;Is it an ESC--purge buffer?
        JNE     NOT_ESC                 ;No, put character into buffer
        CALL    PURGE_BUFFER            ;Yes, remove all characters from buffer
        JMP     READ_LOOP               ;Start reading characters again

NOT_ESC:
        CMP     BL,[SI]                 ;Check to see if buffer is full
        JA      BUFFER_FULL             ;Buffer is full
        MOV     [SI+BX],AL              ;Else save char in buffer
        INC     BX                      ;Point to next free character in buffer
        PUSH    DX
        MOV     DL,AL                   ;Echo character to screen
        CALL    WRITE_CHAR
        POP     DX
        JMP     READ_LOOP               ; Read the next character

;-----------------------------------------;
; Signal an error condition by sending a beep  ;
; character to the display: chr$(7).          ;
;-----------------------------------------;
SIGNAL_ERROR:
        PUSH    DX
        MOV     DL,7                    ;Sound the bell by writing chr$(7)
        MOV     AH,2
        INT     21h
        POP     DX
        JMP     SHORT READ_LOOP         ;Now read next character

;-----------------------------------------;
; The buffer was full, so can't read another   ;
; character.  Send a beep to alert user of     ;
; buffer-full condition.                       ;
;-----------------------------------------;
BUFFER_FULL:
        JMP     SHORT SIGNAL_ERROR      ;If buffer full, just beep

;-----------------------------------------;
; Read the extended ASCII code and place this  ;
; in the buffer as the only character, then    ;
; return -1 as the number of characters read.  ;
;-----------------------------------------;
```

```
EXTENDED:                                       ;Read an extended ASCII code
        CALL    PURGE_BUFFER                    ;Remove any chars from buffer
        MOV     [SI+2],AL                       ;Place just this char in buffer
        MOV     BL,0FFh                         ;Num chars read = -1 for special
        JMP     SHORT END_STRING

;-----------------------------------------------;
; Save the count of the number of characters    ;
; read and return.                              ;
;-----------------------------------------------;
END_INPUT:                                      ;Done with input
        SUB     BL,2                            ;Count of characters read
END_STRING:
        MOV     [SI+1],BL                       ;Return number of chars read
        POP     SI
        POP     BX
        POP     AX
        RET
READ_STRING     ENDP


;----------------------------------------------------------------------;
; This subroutine is used by READ_STRING to clear the contents of the  ;
; input buffer.                                                        ;
;                                                                      ;
;       DS:SI   Points to the input buffer for READ_STRING             ;
;----------------------------------------------------------------------;
PURGE_BUFFER    PROC
        PUSH    CX
        MOV     CL,[SI]                         ;Backspace over maximum number of
        XOR     CH,CH
PURGE_LOOP:                                     ; characters in buffer.  BACK_SPACE
        CALL    BACK_SPACE                      ; will keep the cursor from moving too
        LOOP    PURGE_LOOP                      ; far back
        POP     CX
        RET
PURGE_BUFFER    ENDP


        PUBLIC  BACK_SPACE
        EXTRN   WRITE_CHAR:PROC
        EXTRN   UPDATE_REAL_CURSOR:PROC
        EXTRN   UPDATE_VIRTUAL_CURSOR:PROC
```

```
;-----------------------------------------------------------------------;
; This procedure deletes characters, one at a time, from the buffer    ;
; and the screen when the buffer is not empty.  BACK_SPACE simply      ;
; returns when the buffer is empty.                                     ;
;                                                                       ;
; On entry:     DS:SI+BX       Most recent character still in buffer  ;
; Returns:      DS:SI+BX       Points to next most recent character   ;
;                                                                       ;
; Uses:         WRITE_CHAR                                              ;
;-----------------------------------------------------------------------;
BACK_SPACE      PROC                        ;Delete one character
        PUSH    AX
        PUSH    DX
        CMP     BX,2                        ;Is buffer empty?
        JE      END_BS                      ;Yes, read the next character
        DEC     BX                          ;Remove one character from buffer
        MOV     AH,2                        ;Remove character from screen
        MOV     DL,BS
        INT     21h
        CALL    UPDATE_VIRTUAL_CURSOR       ;Update ptrs after moving real cursor
        MOV     DL,20h                      ;Write space there
        CALL    WRITE_CHAR                  ;Write, but don't move cursor
        CALL    UPDATE_REAL_CURSOR          ;Cursor moved, update real position
        MOV     DL,BS                       ;Back up again
        INT     21h
        CALL    UPDATE_VIRTUAL_CURSOR       ;Set points back to cursor position
END_BS: POP     DX
        POP     AX
        RET
BACK_SPACE      ENDP


        PUBLIC  READ_BYTE
;-----------------------------------------------------------------------;
; This procedure reads either a single ASCII character or a two-digit  ;
; hex number.  This is just a test version of READ_BYTE.               ;
;                                                                       ;
; Returns:      AL      Character code (unless AH = 0)                 ;
;               AH      0 if read ASCII char                           ;
;                       1 if read a special key                        ;
;                       -1 if no characters read                       ;
;                                                                       ;
; Uses:         HEX_TO_BYTE, STRING_TO_UPPER, READ_STRING              ;
; Reads:        KEYBOARD_INPUT, etc.                                    ;
; Writes:       KEYBOARD_INPUT, etc.                                    ;
;-----------------------------------------------------------------------;
```

```
READ_BYTE       PROC
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,3        ;Allow only two characters (plus Enter)
        LEA     DX,KEYBOARD_INPUT
        CALL    READ_STRING
        CMP     NUM_CHARS_READ,1        ;See how many characters
        JE      ASCII_INPUT             ;Just one, treat as ASCII character
        JB      NO_CHARACTERS           ;Only Enter key hit
        CMP     BYTE PTR NUM_CHARS_READ,0FFh    ;Special function key?
        JE      SPECIAL_KEY             ;Yes
        CALL    STRING_TO_UPPER         ;No, convert string to uppercase
        LEA     DX,CHARS                ;Address of string to convert
        CALL    HEX_TO_BYTE             ;Convert string from hex to byte
        JC      NO_CHARACTERS           ;Error, so return 'no characters read'
        XOR     AH,AH                   ;Signal read one byte
DONE_READ:
        POP     DX
        RET
NO_CHARACTERS:
        XOR     AH,AH                   ;Set to 'no characters read'
        NOT     AH                      ;Return -1 in AH
        JMP     DONE_READ
ASCII_INPUT:
        MOV     AL,CHARS                ;Load character read
        XOR     AH,AH                   ;Signal read one byte
        JMP     DONE_READ
SPECIAL_KEY:
        MOV     AL,CHARS[0]             ;Return the scan code
        MOV     AH,1                    ;Signal special key with 1
        JMP     DONE_READ
READ_BYTE       ENDP


        PUBLIC  READ_KEY
;-----------------------------------------------------------------------;
; This procedure reads one key from the keyboard.                       ;
;                                                                       ;
; Returns:      AL      Character code (unless AH = 1)                  ;
;               AH      0 if read ASCII char                            ;
;                       1 if read a special key                         ;
;-----------------------------------------------------------------------;
READ_KEY        PROC
        XOR     AH,AH                   ;Ask for keyboard read function
        INT     16h                     ;Read character/scan code from keyboard
        OR      AL,AL                   ;Is it an extended code?
        JZ      EXTENDED_CODE           ;Yes
```

```
NOT_EXTENDED:
        XOR     AH,AH                   ;Return just the ASCII code
DONE_READING:
        RET

EXTENDED_CODE:
        MOV     AL,AH                   ;Put scan code into AL
        MOV     AH,1                    ;Signal extended code
        JMP     DONE_READING
READ_KEY        ENDP


        PUBLIC  READ_DECIMAL
;-----------------------------------------------------------------------;
; This procedure takes the output buffer of READ_STRING and converts    ;
; the string of decimal digits to a word.                               ;
;                                                                       ;
; Returns:      AX      Word converted from decimal                     ;
;               CF      Set if error, clear if no error                 ;
;                                                                       ;
; Uses:         READ_STRING                                             ;
; Reads:        KEYBOARD_INPUT, etc.                                    ;
; Writes:       KEYBOARD_INPUT, etc.                                    ;
;-----------------------------------------------------------------------;
READ_DECIMAL    PROC
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     CHAR_NUM_LIMIT,6        ;Max number is 5 digits (65535)
        LEA     DX,KEYBOARD_INPUT
        CALL    READ_STRING
        MOV     CL,NUM_CHARS_READ       ;Get number of characters read
        XOR     CH,CH                   ;Set upper byte of count to 0
        CMP     CL,0                    ;Return error if no characters read
        JLE     BAD_DECIMAL_DIGIT       ;No chars read, signal error
        XOR     AX,AX                   ;Start with number set to 0
        XOR     BX,BX                   ;Start at beginning of string
CONVERT_DIGIT:
        MOV     DX,10                   ;Multiply number by 10
        MUL     DX                      ;Multiply AX by 10
        JC      BAD_DECIMAL_DIGIT       ;CF set if MUL overflowed one word
        MOV     DL,CHARS[BX]            ;Get the next digit
        SUB     DL,'0'                  ;And convert to a nibble (4 bits)
        JS      BAD_DECIMAL_DIGIT          ;Bad digit if < 0
        CMP     DL,9                       ;Is this a bad digit?
        JA      BAD_DECIMAL_DIGIT          ;Yes
        ADD     AX,DX                      ;No, so add it to number
        INC     BX                         ;Point to next character
        LOOP    CONVERT_DIGIT              ;Get the next digit
DONE_DECIMAL:
        POP     DX
        POP     CX
        POP     BX
        RET
BAD_DECIMAL_DIGIT:
        STC                                ;Set carry to signal error
        JMP     DONE_DECIMAL
READ_DECIMAL    ENDP


        END
```

PHANTOM.ASM

```
.MODEL  SMALL

.DATA

REAL_CURSOR_X           DB      0
REAL_CURSOR_Y           DB      0
        PUBLIC  PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y
PHANTOM_CURSOR_X        DB      0
PHANTOM_CURSOR_Y        DB      0

.CODE

;-----------------------------------------------------------------------;
; These four procedures move the phantom cursors.                       ;
;                                                                       ;
; Uses:         ERASE_PHANTOM, WRITE_PHANTOM                            ;
;               SCROLL_DOWN, SCROLL_UP                                  ;
; Reads:        PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y                      ;
; Writes:       PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y                      ;
;-----------------------------------------------------------------------;
```

```
        PUBLIC  PHANTOM_UP
PHANTOM_UP      PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        DEC     PHANTOM_CURSOR_Y        ;Move cursor up one line
        JNS     WASNT_AT_TOP            ;Was not at the top, write cursor
        CALL    SCROLL_DOWN             ;Was at the top, scroll
WASNT_AT_TOP:
        CALL    WRITE_PHANTOM           ;Write the phantom at new position
        RET
PHANTOM_UP      ENDP

        PUBLIC  PHANTOM_DOWN
PHANTOM_DOWN    PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        INC     PHANTOM_CURSOR_Y        ;Move cursor down one line
        CMP     PHANTOM_CURSOR_Y,16     ;Was it at the bottom?
        JB      WASNT_AT_BOTTOM         ;No, so write phantom
        CALL    SCROLL_UP               ;Was at bottom, scroll
WASNT_AT_BOTTOM:
        CALL    WRITE_PHANTOM           ;Write the phantom cursor
        RET
PHANTOM_DOWN    ENDP

        PUBLIC  PHANTOM_LEFT
PHANTOM_LEFT    PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        DEC     PHANTOM_CURSOR_X        ;Move cursor left one column
        JNS     WASNT_AT_LEFT           ;Was not at the left side, write cursor
        MOV     PHANTOM_CURSOR_X,0      ;Was at left, so put back there
WASNT_AT_LEFT:
        CALL    WRITE_PHANTOM           ;Write the phantom cursor
        RET
PHANTOM_LEFT    ENDP

        PUBLIC  PHANTOM_RIGHT
PHANTOM_RIGHT   PROC
        CALL    ERASE_PHANTOM           ;Erase at current position
        INC     PHANTOM_CURSOR_X        ;Move cursor right one column
        CMP     PHANTOM_CURSOR_X,16     ;Was it already at the right side?
        JB      WASNT_AT_RIGHT
        MOV     PHANTOM_CURSOR_X,15     ;Was at right, so put back there
WASNT_AT_RIGHT:
        CALL    WRITE_PHANTOM           ;Write the phantom cursor
        RET
PHANTOM_RIGHT   ENDP
```

```
        PUBLIC  MOV_TO_HEX_POSITION
        EXTRN   GOTO_XY:PROC
.DATA
        EXTRN   LINES_BEFORE_SECTOR:BYTE
.CODE
;-----------------------------------------------------------------------;
; This procedure moves the real cursor to the position of the phantom  ;
; cursor in the hex window.                                             ;
;                                                                       ;
; Uses:         GOTO_XY                                                 ;
; Reads:        LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y;
;-----------------------------------------------------------------------;
MOV_TO_HEX_POSITION     PROC
        PUSH    AX
        PUSH    CX
        PUSH    DX
        MOV     DH,LINES_BEFORE_SECTOR  ;Find row of phantom (0,0)
        ADD     DH,2                    ;Plus row of hex and horizontal bar
        ADD     DH,PHANTOM_CURSOR_Y     ;DH = row of phantom cursor
        MOV     DL,8                    ;Indent on left side
        MOV     CL,3                    ;Each column uses 3 characters, so
        MOV     AL,PHANTOM_CURSOR_X     ; we must multiply CURSOR_X by 3
        MUL     CL
        ADD     DL,AL                   ;And add to the indent, to get column
        CALL    GOTO_XY                 ; for phantom cursor
        POP     DX
        POP     CX
        POP     AX
        RET
MOV_TO_HEX_POSITION     ENDP

        PUBLIC  MOV_TO_ASCII_POSITION
        EXTRN   GOTO_XY:PROC
.DATA
        EXTRN   LINES_BEFORE_SECTOR:BYTE
.CODE
;-----------------------------------------------------------------------;
; This procedure moves the real cursor to the beginning of the phantom ;
; cursor in the ASCII window.                                           ;
;                                                                       ;
; Uses:         GOTO_XY                                                 ;
; Reads:        LINES_BEFORE_SECTOR, PHANTOM_CURSOR_X, PHANTOM_CURSOR_Y;
;-----------------------------------------------------------------------;
```

```
MOV_TO_ASCII_POSITION   PROC
        PUSH    AX
        PUSH    DX
        MOV     DH,LINES_BEFORE_SECTOR  ;Find row of phantom (0,0)
        ADD     DH,2                    ;Plus row of hex and horizontal bar
        ADD     DH,PHANTOM_CURSOR_Y     ;DH = row of phantom cursor
        MOV     DL,59                   ;Indent on left side
        ADD     DL,PHANTOM_CURSOR_X     ;Add CURSOR_X to get X position
        CALL    GOTO_XY                 ; for phantom cursor
        POP     DX
        POP     AX
        RET
MOV_TO_ASCII_POSITION   ENDP

        PUBLIC  SAVE_REAL_CURSOR
;-----------------------------------------------------------------------;
; This procedure saves the position of the real cursor in the two      ;
; variables REAL_CURSOR_X and REAL_CURSOR_Y.                            ;
;                                                                       ;
; Writes:       REAL_CURSOR_X, REAL_CURSOR_Y                            ;
;-----------------------------------------------------------------------;
SAVE_REAL_CURSOR        PROC
        PUSH    AX
        PUSH    BX
        PUSH    CX
        PUSH    DX
        MOV     AH,3                    ;Read cursor position
        XOR     BH,BH                   ; on page 0
        INT     10h                     ;And return in DL,DH
        MOV     REAL_CURSOR_Y,DL        ;Save position
        MOV     REAL_CURSOR_X,DH
        POP     DX
        POP     CX
        POP     BX
        POP     AX
        RET
SAVE_REAL_CURSOR        ENDP

        PUBLIC  RESTORE_REAL_CURSOR
        EXTRN   GOTO_XY:PROC
;-----------------------------------------------------------------------;
; This procedure restores the real cursor to its old position, saved   ;
; in REAL_CURSOR_X and REAL_CURSOR_Y.                                   ;
;                                                                       ;
; Uses:         GOTO_XY                                                 ;
; Reads:        REAL_CURSOR_X, REAL_CURSOR_Y                            ;
;-----------------------------------------------------------------------;
```

```
RESTORE_REAL_CURSOR     PROC
        PUSH    DX
        MOV     DL,REAL_CURSOR_Y
        MOV     DH,REAL_CURSOR_X
        CALL    GOTO_XY
        POP     DX
        RET
RESTORE_REAL_CURSOR     ENDP

        PUBLIC  WRITE_PHANTOM
        EXTRN   WRITE_ATTRIBUTE_N_TIMES:PROC
;-----------------------------------------------------------------------;
; This procedure uses CURSOR_X and CURSOR_Y, through MOV_TO_..., as the;
; coordinates for the phantom cursor.  WRITE_PHANTOM writes this       ;
; phantom cursor.                                                       ;
;                                                                       ;
; Uses:         WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR               ;
;               RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION                ;
;               MOV_TO_ASCII_POSITION                                   ;
;-----------------------------------------------------------------------;
WRITE_PHANTOM   PROC
        PUSH    CX
        PUSH    DX
        CALL    SAVE_REAL_CURSOR
        CALL    MOV_TO_HEX_POSITION     ;Coord. of cursor in hex window
        MOV     CX,4                    ;Make phantom cursor four chars wide
        MOV     DL,70h
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    MOV_TO_ASCII_POSITION   ;Coord. of cursor in ASCII window
        MOV     CX,1                    ;Cursor is one character wide here
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    RESTORE_REAL_CURSOR
        POP     DX
        POP     CX
        RET
WRITE_PHANTOM   ENDP

        PUBLIC  ERASE_PHANTOM
        EXTRN   WRITE_ATTRIBUTE_N_TIMES:PROC
;-----------------------------------------------------------------------;
; This procedure erases the phantom cursor, just the opposite of       ;
; WRITE_PHANTOM.                                                        ;
;                                                                       ;
; Uses:         WRITE_ATTRIBUTE_N_TIMES, SAVE_REAL_CURSOR               ;
;               RESTORE_REAL_CURSOR, MOV_TO_HEX_POSITION                ;
;               MOV_TO_ASCII_POSITION                                   ;
;-----------------------------------------------------------------------;
```

```
ERASE_PHANTOM   PROC
        PUSH    CX
        PUSH    DX
        CALL    SAVE_REAL_CURSOR
        CALL    MOV_TO_HEX_POSITION     ;Coord. of cursor in hex window
        MOV     CX,4                    ;Change back to white on black
        MOV     DL,7
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    MOV_TO_ASCII_POSITION
        MOV     CX,1
        CALL    WRITE_ATTRIBUTE_N_TIMES
        CALL    RESTORE_REAL_CURSOR
        POP     DX
        POP     CX
        RET
ERASE_PHANTOM   ENDP


        EXTRN   DISP_HALF_SECTOR:PROC, GOTO_XY:PROC
.DATA
        EXTRN   SECTOR_OFFSET:WORD
        EXTRN   LINES_BEFORE_SECTOR:BYTE
.CODE
;-----------------------------------------------------------------------;
; These two procedures move between the two half-sector displays.      ;
;                                                                       ;
; Uses:         WRITE_PHANTOM, DISP_HALF_SECTOR, ERASE_PHANTOM, GOTO_XY;
;               SAVE_REAL_CURSOR, RESTORE_REAL_CURSOR                   ;
; Reads:        LINES_BEFORE_SECTOR                                     ;
; Writes:       SECTOR_OFFSET, PHANTOM_CURSOR_Y                         ;
;-----------------------------------------------------------------------;
SCROLL_UP       PROC
        PUSH    DX
        CALL    ERASE_PHANTOM           ;Remove the phantom cursor
        CALL    SAVE_REAL_CURSOR        ;Save the real cursor position
        XOR     DL,DL                   ;Set cursor for half-sector display
        MOV     DH,LINES_BEFORE_SECTOR
        ADD     DH,2
        CALL    GOTO_XY
        MOV     DX,256                  ;Display the second half sector
        MOV     SECTOR_OFFSET,DX
        CALL    DISP_HALF_SECTOR
        CALL    RESTORE_REAL_CURSOR     ;Restore the real cursor position
        MOV     PHANTOM_CURSOR_Y,0      ;Cursor at top of second half sector
        CALL    WRITE_PHANTOM           ;Restore the phantom cursor
        POP     DX
        RET
SCROLL_UP       ENDP

SCROLL_DOWN     PROC
        PUSH    DX
        CALL    ERASE_PHANTOM           ;Remove the phantom cursor
        CALL    SAVE_REAL_CURSOR        ;Save the real cursor position
        XOR     DL,DL                   ;Set cursor for half-sector display
        MOV     DH,LINES_BEFORE_SECTOR
        ADD     DH,2
        CALL    GOTO_XY
        XOR     DX,DX                   ;Display the first half sector
        MOV     SECTOR_OFFSET,DX
        CALL    DISP_HALF_SECTOR
        CALL    RESTORE_REAL_CURSOR     ;Restore the real cursor position
        MOV     PHANTOM_CURSOR_Y,15     ;Cursor at bottom of first half sector
        CALL    WRITE_PHANTOM           ;Restore the phantom cursor
        POP     DX
        RET
SCROLL_DOWN     ENDP


        END
```

VIDEO_IO.ASM

```
.MODEL  SMALL

.DATA
        PUBLIC  SCREEN_PTR
        PUBLIC  SCREEN_X, SCREEN_Y
SCREEN_SEG      DW      0B800h          ;Setment of the screen buffer
SCREEN_PTR      DW      0               ;Offset into screen memory of cursor
SCREEN_X        DB      0               ;Position of the screen cursor
SCREEN_Y        DB      0

.CODE
```

```
        PUBLIC  WRITE_STRING
;----------------------------------------------------------------------;
; This procedure writes a string of characters to the screen.  The     ;
; string must end with      DB      0                                  ;
;                                                                      ;
; On entry:     DS:DX   Address of the string                          ;
;                                                                      ;
; Uses:         WRITE_CHAR                                             ;
;----------------------------------------------------------------------;
WRITE_STRING    PROC
        PUSH    AX
        PUSH    DX
        PUSH    SI
        PUSHF                           ;Save direction flag
        CLD                             ;Set direction for increment (forward)
        MOV     SI,DX                   ;Place address into SI for LODSB
STRING_LOOP:
        LODSB                           ;Get a character into the AL register
        OR      AL,AL                   ;Have we found the 0 yet?
        JZ      END_OF_STRING           ;Yes, we are done with the string
        MOV     DL,AL                   ;No, write character
        CALL    WRITE_CHAR
        JMP     STRING_LOOP
END_OF_STRING:
        POPF                            ;Restore direction flag
        POP     SI
        POP     DX
        POP     AX
        RET
WRITE_STRING    ENDP


        PUBLIC  WRITE_HEX
;----------------------------------------------------------------------;
; This procedure converts the byte in the DL register to hex and writes;
; the two hex digits at the current cursor position.                   ;
;                                                                      ;
; On Entry:     DL      Byte to be converted to hex.                   ;
;                                                                      ;
; Uses: WRITE_HEX_DIGIT                                                ;
;----------------------------------------------------------------------;
WRITE_HEX       PROC                    ;Entry point
        PUSH    CX                      ;Save registers used in this procedure
        PUSH    DX
        MOV     DH,DL                   ;Make a copy of byte
        MOV     CX,4                    ;Get the upper nibble in DL
        SHR     DL,CL
        CALL    WRITE_HEX_DIGIT         ;Display first hex digit
        MOV     DL,DH                   ;Get lower nibble into DL
        AND     DL,0Fh                  ;Remove the upper nibble
        CALL    WRITE_HEX_DIGIT         ;Display second hex digit
        POP     DX
        POP     CX
        RET
WRITE_HEX       ENDP

        PUBLIC  WRITE_HEX_DIGIT
;-----------------------------------------------------------------------;
; This procedure converts the lower 4 bits of DL to a hex digit and    ;
; writes it to the screen.                                              ;
;                                                                       ;
; On Entry:     DL      Lower 4 bits contain number to be printed      ;
;                       in hex.                                         ;
;                                                                       ;
; Uses:         WRITE_CHAR                                              ;
;-----------------------------------------------------------------------;
WRITE_HEX_DIGIT PROC
        PUSH    DX                      ;Save registers used
        CMP     DL,10                   ;Is this nibble <10?
        JAE     HEX_LETTER              ;No, convert to a letter
        ADD     DL,"0"                  ;Yes, convert to a digit
        JMP     Short WRITE_DIGIT       ;Now write this character
HEX_LETTER:
        ADD     DL,"A"-10               ;Convert to hex letter
WRITE_DIGIT:
        CALL    WRITE_CHAR              ;Display the letter on the screen
        POP     DX                      ;Restore old value of DX
        RET
WRITE_HEX_DIGIT ENDP


        PUBLIC  INIT_WRITE_CHAR
;-----------------------------------------------------------------------;
; You need to call this procedure before you call WRITE_CHAR since     ;
; WRITE_CHAR uses information set by this procedure.                   ;
;                                                                       ;
; Writes:       SCREEN_SEG                                              ;
;-----------------------------------------------------------------------;
```

```
INIT_WRITE_CHAR PROC
        PUSH    AX
        PUSH    BX
        MOV     BX,0B800h               ;Set for color graphics display
        INT     11h                     ;Get equipment information
        AND     AL,30h                  ;Keep just the video display type
        CMP     AL,30h                  ;Is this a monochrome display adapter?
        JNE     SET_BASE                ;No, it's color, so use B800
        MOV     BX,0B000h               ;Yes, it's monochrome, so use B000
SET_BASE:
        MOV     SCREEN_SEG,BX           ;Save the screen segment
        POP     BX
        POP     AX
        RET
INIT_WRITE_CHAR ENDP


        PUBLIC  WRITE_CHAR
        EXTRN   CURSOR_RIGHT:PROC
;-------------------------------------------------------------------------;
; This procedure outputs a character to the screen by writing directly ;
; into screen memory, so that characters such as the backspace are     ;
; treated as any other characters and are displayed.                   ;
;                                                                      ;
; This procedure must do a bit of work to update the cursor position.  ;
;                                                                      ;
; On entry:     DL      Byte to print on screen                        ;
;                                                                      ;
; Uses:         CURSOR_RIGHT                                           ;
; Reads:        SCREEN_SEG, SCREEN_PTR                                 ;
;-------------------------------------------------------------------------;
WRITE_CHAR      PROC
        PUSH    AX
        PUSH    BX
        PUSH    DX
        PUSH    ES

        MOV     AX,SCREEN_SEG           ;Get segment for screen memory
        MOV     ES,AX                   ;Point ES to screen memory
        MOV     BX,SCREEN_PTR           ;Pointer to character in screen memory

        MOV     DH,7                    ;Use the normal attribute
        MOV     ES:[BX],DX              ;Write character/attribute to screen
        CALL    CURSOR_RIGHT            ;Now move to next cursor position
        POP     ES
        POP     DX                      ;Restore old value in AX and DX
        POP     BX
        POP     AX
        RET
WRITE_CHAR      ENDP


        PUBLIC  WRITE_DECIMAL
;-----------------------------------------------------------------------;
; This procedure writes a 16-bit, unsigned number in decimal notation. ;
;                                                                       ;
; On Entry:     DX      N : 16-bit, unsigned number.                    ;
;                                                                       ;
; Uses:         WRITE_HEX_DIGIT                                         ;
;-----------------------------------------------------------------------;
WRITE_DECIMAL   PROC
        PUSH    AX                      ;Save registers used here
        PUSH    CX
        PUSH    DX
        PUSH    SI
        MOV     AX,DX
        MOV     SI,10                   ;Will divide by 10 using SI
        XOR     CX,CX                   ;Count of digits placed on stack
NON_ZERO:
        XOR     DX,DX                   ;Set upper word of N to 0
        DIV     SI                      ;Calculate N/10 and (N mod 10)
        PUSH    DX                      ;Push one digit onto the stack
        INC     CX                      ;One more digit added
        OR      AX,AX                   ;N = 0 yet?
        JNE     NON_ZERO                ;Nope, continue
WRITE_DIGIT_LOOP:
        POP     DX                      ;Get the digits in reverse order
        CALL    WRITE_HEX_DIGIT
        LOOP    WRITE_DIGIT_LOOP
END_DECIMAL:
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
WRITE_DECIMAL   ENDP
```

```
        PUBLIC  WRITE_CHAR_N_TIMES
;----------------------------------------------------------------------;
; This procedure writes more than one copy of a character             ;
;                                                                      ;
; On entry:     DL      Character code                                 ;
;               CX      Number of times to write the character         ;
;                                                                      ;
; Uses:         WRITE_CHAR                                             ;
;----------------------------------------------------------------------;
WRITE_CHAR_N_TIMES      PROC
        PUSH    CX
N_TIMES:
        CALL    WRITE_CHAR
        LOOP    N_TIMES
        POP     CX
        RET
WRITE_CHAR_N_TIMES   ENDP

      PUBLIC WRITE ATTRIBUTE N TIMES
     EXTRN CURSOR RIGHT:PROC
;----------------------------------------------------------------------;
; This procedure sets the attribute for N characters, starting at the ;
; current cursor position.                                             ;
;                                                                      ;
; On entry:     CX      Number of characters to set attribute for      ;
;               DL      New attribute for characters                   ;
;                                                                      ;
; Uses:         CURSOR_RIGHT                                           ;
;----------------------------------------------------------------------;
WRITE_ATTRIBUTE_N_TIMES PROC
        PUSH    AX
        PUSH    CX
        PUSH    DI
        PUSH    ES

        MOV     AX,SCREEN_SEG           ;Set ES to point to screen segment
        MOV     ES,AX
        MOV     DI,SCREEN_PTR           ;Character under cursor
        INC     DI                      ;Point to the attribute under cursor
        MOV     AL,DL                   ;Put attribute into AL
ATTR_LOOP:
        STOSB                           ;Save one attribute
        INC     DI                      ;Move to next attribute
        INC     SCREEN_X                ;Move to next column
        LOOP    ATTR_LOOP               ;Write N attributes
        DEC     DI                              ;Point to start of next character
        MOV     SCREEN_PTR,DI                   ;Remember where we are

        POP     ES
        POP     DI
        POP     CX
        POP     AX
        RET
WRITE_ATTRIBUTE_N_TIMES ENDP


        PUBLIC  WRITE_PATTERN
;----------------------------------------------------------------------;
; This procedure writes a line to the screen, based on data in the     ;
; form                                                                 ;
;                                                                      ;
;       DB      {character, number of times to write character}, 0    ;
; Where {x} means that x can be repeated any number of times           ;
;                                                                      ;
; On entry:     DS:DX   Address of the pattern to draw                 ;
;                                                                      ;
; Uses:         WRITE_CHAR_N_TIMES                                     ;
;----------------------------------------------------------------------;
WRITE_PATTERN   PROC
        PUSH    AX
        PUSH    CX
        PUSH    DX
        PUSH    SI
        PUSHF                                   ;Save the direction flag
        CLD                                     ;Set direction flag for increment
        MOV     SI,DX                           ;Move offset into SI register for LODSB
PATTERN_LOOP:
        LODSB                                   ;Get character data into AL
        OR      AL,AL                           ;Is it the end of data (0h)?
        JZ      END_PATTERN                     ;Yes, return
        MOV     DL,AL                           ;No, set up to write character N times
        LODSB                                   ;Get the repeat count into AL
        MOV     CL,AL                           ;And put in CX for WRITE_CHAR_N_TIMES
        XOR     CH,CH                           ;Zero upper byte of CX
        CALL    WRITE_CHAR_N_TIMES
        JMP     PATTERN_LOOP
```

```
END_PATTERN:
        POPF                                    ;Restore direction flag
        POP     SI
        POP     DX
        POP     CX
        POP     AX
        RET
WRITE_PATTERN   ENDP


        END
```

A P P E N D I X

C/C++ Libraries in Assembly

This appendix contains the source listings for the C/C++ libraries mentioned in Chapter 31. These files are included so that you can build a library that will work with any memory model. All you have to change is a single line in the Makefile. Although the assembled library can be used with most C/C++ compilers, you will need MASM 6 to build the CLIB.LIB file.

Topics Covered

Descriptions of Procedures

Descriptions of Procedures

The CLIB directory on the disk included with this book contains five assembly files that include procedures and functions written for C/C++ programs. Using the methods discussed in Chapter 31, calls to the functions can be added to C/C++ files to do fast writes to the screen, change cursor properties, and move a mouse cursor. The included makefile allows you to compile the files into a library called CLIB.LIB that can be linked to your C/C++ files. You will need MASM 6 to assemble the files.

SOCHALIB.INC

This is an include file used by all the other ASM files.

MAKEFILE

This makefile is used to create the CLIB.LIB file.

CURSOR.ASM

The seven functions and procedures in this module control the hardware cursor you see on the screen.

void SetCursor(int size) This procedure allows you to set the size of the hardware cursor. Normally you would use this function along with GetVisibleCursor or GetCursor to save and restore the size of the cursor. For example, at the start of your program you should call GetVisibleCursor and save the return value in a global variable. Then call SetCursor with this value at the end of your program to restore the DOS cursor before your program exits.

int GetVisibleCursor(void) This function returns the size of the hardware cursor. If the cursor is turned off, such as by using CursorOff, this function returns a value that would be visible.

int GetCursor(void) This function is like GetVisibleCursor except that it returns the exact size of the cursor. If the cursor is turned off, you will get a cursor size that will turn the cursor off when you use this value with SetCursor.

void CursorOff(void) This procedure hides the hardware cursor. You can call CursorOn to turn the cursor back on. You cannot nest calls to CursorOff and CursorOn because they use a single value to save the previous state of the cursor.

void CursorOn(void) This procedure turns the cursor back on after you call CursorOff to hide the hardware cursor. You cannot nest calls to CursorOff and CursorOn because they use a single value to save the previous state of the cursor.

void CursorBlock(void) This procedure turns the hardware cursor into a blinking block cursor. This cursor will be the full height of characters. A block cursor is usually easier to see than the normal underscore cursor.

void CursorUnderscore(void) This procedure turns the cursor into the normal underscore cursor.

FASTWRIT.ASM

The 12 procedures in this module support writing characters very quickly to your screen. These procedures also work together with the procedures in MOUSE.ASM to make sure writing to the screen does not interfere with the mouse cursor. In other words, these routines hide the mouse cursor before writing to the screen, so you do not have to worry about hiding the mouse cursor yourself. If you do not hide the mouse cursor, you can write over the mouse cursor, leading to "mouse droppings" when you move the mouse.

void InitFastDisplayModule(void) Call this procedure to initialize the entire module. You will need to call this procedure before you call any other procedures in this module.

void FastFlush(void) Flushes any characters that have not been written to the screen. The procedures in this module put characters in an off-screen buffer, which makes the code extremely fast. You will usually want to call fast flush during your keyboard input code. In fact, you might want to insert calls to FastFlush in KBD_IO.ASM.

void FastWriteRawChar(char c) Writes a single character to the screen without special meaning. The other procedure, FastWriteChar, treats some characters as special characters, such as the tab character. But this procedure displays all characters as symbols on the screen rather than giving them special meaning.

void FastWriteChar(char c) Like FastWriteRawChar, this procedure writes a character to the screen. However, it gives special meaning to the following characters: ASCII 7 produces a beep; ASCII 9 is the tab character and moves the cursor to the next tab stop; ASCII 10 moves the cursor down one line; and ASCII 13 moves the cursor to the start of the current line.

int FastReadAttr(void) Returns the character attribute for the character currently under the cursor.

void FastWriteString(char *s) Displays the C-style string on the screen. This procedure uses FastWriteChar to display each character in the string.

void FastWriteUDecimal(int i) Displays the number *i* on the screen as an unsigned decimal number. In other words, the numbers range from 0 to 65535.

void FastWriteNChars(char c, int count) Displays *count* copies of the character *c* on the screen. This procedure uses FastWriteRawChar to display the individual characters on the screen.

void FastWriteSpaces(int count) Displays *count* spaces on the screen.

void FastGotoXY(int x, int y) Moves the cursor to (x, y). This procedure moves both the virtual cursor in FASTWRIT.ASM and the hardware cursor to the new location. You can use CursorOff and CursorOn to hide the hardware cursor when you draw on the screen.

void FastSetCursor(void) Moves the hardware cursor so it matches the position of the virtual cursor. Normally the hardware cursor's position won't be updated after a call to FastFlush.

void FastGetXY(int *x, int *y) Returns the position of the current cursor in (x, y).

HARDWARE.ASM

The three functions in this file report information about your display hardware.

int GetDisplayType(void) Returns information on the type of display card in your computuer: 0 = no cards, 1 = color 40 x 25 card, 2 = color 80 x 25 card, and 3 = monochrome-display card.

int EgaActive(void) Returns true (–1) if you have an EGA or VGA card in your computer and it is currently the active display.

int GetDisplayRows(void) Returns the number of rows on the screen (normally 25, but can be higher numbers).

KBD_IO.ASM

The five functions in this file handle input from the keyboard.

int ReadKey(void) This function uses the ROM BIOS routines to read a single character. It returns the ASCII code for keys that generate an ASCII character. If you push one of the key-pad or function keys, this function returns the scan code in the lower byte and sets the upper byte to 1.

int ReadRawKey(void) This procedure reads one character (or one special function key) from the keyboard and returns the ASCII code, or scan code. It uses the ROM BIOS call so it can read both the character code AND the scan code. The DOS call returns only the ASCII code. On return, the scan code is in the upper byte and the ASCII code (or 0 for non-ASCII keys) is in the lower byte.

void ClearKbdBuffer(void) Clears any characters that are currently in the keyboard buffer. It is a good idea to call this procedure immediately after an error occurs to keep the user from accidently typing the wrong characters.

int KeyWaiting(void) Reports if any characters are waiting to be read from the keyboard buffer, and if so, it returns the same character code that ReadRawChar would return; otherwise, it returns –1.

int ShiftKeys(void) Reports which shift keys are currently down:

80h	Insert state on
40h	Caps Lock on
20h	Num Lock on
10h	Scroll Lock on
08h	Alt key is down
04h	Control key is down
02h	Left shift key is down
01h	Right shift key is down

MOUSE.ASM

The procedures in this module support using a mouse in DOS programs. These procedures are designed to work along with the procedures in FASTWRIT.ASM, but you can easily rewrite them to work without FASTWRIT.ASM. If you do, keep in mind that you will need to hide the mouse cursor before you draw on the screen.

void InitMouse(void) Call this procedure before you call any of the other procedures in this module. This procedure initializes the mouse by calling function 0, which is a full reset of the mouse software AND hardware. In some cases this can take several seconds (a serial mouse or an IBM mouse), which may not be acceptable. Init_mousesw( ) can be much faster in such cases.

The following rules state which of these two procedures to call:

- Once   Init_mouse( ) must be called at least to initialize the mouse hardware the first time because not all mouse drivers initialize the mouse properly when you do a software mouse reset.
- After Init_mouse( )   From then on you can call Init_mouse_sw( ) for a fast reset. The Commander calls Init_mouse( ) once, then Init_mouse_sw( ) each time you return to the Commander after running a DOS command.

void InitMouseSw(void) This procedure initializes the mouse. It tries to do a software reset first. Only if the software reset doesn't work does it try to do a hardware reset. In some cases (such as a serial mouse or the IBM PS/2 mouse) a hardware reset can take as much as five seconds, which is not an acceptable delay when returning from the Norton Commander.

void UnhideMouse(void) Makes the mouse visible again after a call to HideMouse. You can make nested calls to UnhideMouse and HideMouse.

void HideMouse(void) Removes the mouse cursor from the screen. You can make nested calls to UnhideMouse and HideMouse.

void GetMousePosition(int *x, int *y) Returns the current position of the mouse cursor in character coordinates.

void SetMousePosition(int x, int y) Allows you to set the position of the mouse on the screen, in character coordinates.

int MouseButtons(int *x, int *y) Like GetMousePosition, it returns the current position of the mouse on the screen. It also reports which mouse buttons were pressed:

0	Both buttons up
1	Left button down
2	Right button down
3	Both buttons down

Makefile

Following is the makefile for creating the libraries:

```
#
# This makefile creates the "Socha Libraries".  To change the memory model,
# change the defintion below of MEM_MODEL to one of the valid memory models.
#
MEM_MODEL = LARGE
```

```
#
# The two lines below define how to build an obj file from an asm file.
# I've used the /D switch to define MEM_MODEL, which is used in the .MODEL
# statement to define the memory model to build.  And /Zi includes the
# debugger information in the file.
#
.asm.obj:
        ml /DMEM_MODEL=$(MEM_MODEL) /c /Zi $<
        lib clib -+$*;

#
# This line causes all the files to be assembled and added to the library.
#
clib.lib:       hardware.obj mouse.obj fastwrit.obj cursor.obj kbd_io.obj
```

SOCHALIB.INC

This file is used by all the ASM files in this appendix.

```
;-----------------------------------------------------------------------;
; I use these macros to write code that works in all memory models.    ;
;-----------------------------------------------------------------------;
        if      @DataSize               ;Are pointers Far?
lodDS   textequ <lds>                   ;Yes, use LDS to get DS:pointer
lodES   textequ <les>                   ;And use LES to get ES:pointer
refES   textequ <es:>                   ;And put ES: in front of refs
        else
lodDS   textequ <mov>                   ;No, use MOV to get pointer
lodES   textequ <mov>                   ;And also for ES case
refES   textequ <>                      ;And nothing for the references
        endif

        if @CodeSize                    ;Are calls Far?
procRef textequ <far>                   ;Yes, use Far
        else
procRef textequ <near>                  ;No, use Near
        endif
.
```

CURSOR.ASM

```
.MODEL  MEM_MODEL,C

INCLUDE sochalib.inc

;-----------------------------------------------------------------------;
; Copyright © 1992 by John Socha                                        ;
;-----------------------------------------------------------------------;

;-----------------------------------------------------------------------;
; This file contains a number of procedures that work with the screen   ;
; cursor.                                                               ;
;                                                                       ;
;       SetCursor               Sets the size of the hardware cursor    ;
;       GetVisibleCursor        Gets the size of a visible cursor       ;
;       GetCursor               Gets the current size of the cursor     ;
;       CursorOff               Turns the cursor off                    ;
;       CursorOn                Turns the cursor back on                ;
;       CursorBlock             Changes the cursor to a block cursor    ;
;       CursorUnderscore        Changes the cursor to an underscore     ;
;                                                                       ;
;       goto_xy                 Moves the cursor position               ;
;       get_xy                  Returns the current cursor position     ;
;       get_active_page         Returns the number of the active page   ;
;       set_cursor              Sets the size of the hardware cursor    ;
;       fix_cursor (private)    Makes sure the cursor size is valid     ;
;-----------------------------------------------------------------------;

.CODE

;-----------------------------------------------------------------------;
; This procedure moves the cursor to another position on the screen.    ;
; It doesn't do any error checking in case you want to do something     ;
; off the screen.  But be careful!                                      ;
;                                                                       ;
; On Entry:     DH      Row (Y)                                         ;
;               DL      Column (X)                                      ;
;                                                                       ;
; Uses:         GET_ACTIVE_PAGE                                         ;
;-----------------------------------------------------------------------;
```

```
goto_xy         proc    uses ax bx si di bp
        call    get_active_page         ;Get the number of the active page
        mov     bh,al                   ;Put this page number into BH
        mov     ah,2                    ;Call for SET CURSOR POSITION
        int     10h
        ret
goto_xy         endp


;-------------------------------------------------------------------;
; This procedure is the reverse of GOTO_XY.  Instead of moving the  ;
; cursor to a new position, it returns the current cursor position. ;
;                                                                   ;
; Returns:      AH      Row (Y)                                     ;
;               AL      Column (X)                                  ;
;                                                                   ;
; Uses:         GET_ACTIVE_PAGE                                     ;
;-------------------------------------------------------------------;
get_xy  proc    uses bx dx si di bp
        call    get_active_page         ;Return active page in AL
        mov     bh,al                   ;Put this page number into BH
        mov     ah,3                    ;Call for read cursor position
        int     10h
        mov     ax,dx
        ret
get_xy  endp


;-------------------------------------------------------------------;
; This procedure gets the page number for the page shown on the screen ;
; and returns it in the AL register.                                ;
;                                                                   ;
; Returns:      AL      Active page number                          ;
;-------------------------------------------------------------------;
get_active_page         proc    uses bx si di bp
        mov     ah,15                   ;Ask for current video state
        int     10h
        mov     al,bh                   ;Put active page number in AL
        ret
get_active_page         endp
```

```
;---------------------------------------------------------------------;
; This variable is used by CursorOn and CursorOff to keep track of the ;
; size of the cursor.                                                 ;
;---------------------------------------------------------------------;
.DATA
old_cursor_type         dw      0       ;Old start and end scan lines
.CODE


;---------------------------------------------------------------------;
; This procedure allows you to set the cursor to a new size.  Usually ;
; you'll use this to restore a cursor that you obtained with          ;
; GetVisibleCursor().                                                 ;
;                                                                     ;
;       void SetCursor(int size);                                     ;
;---------------------------------------------------------------------;
SetCursor       proc    cursorSize:Word
        mov     dx, cursorSize
        call    set_cursor
        ret
SetCursor       endp


;---------------------------------------------------------------------;
; This procedure sets the end and start lines of the cursor.          ;
;                                                                     ;
; On entry:     DH      First scan line of cursor.                    ;
;               DL      Last scan line of cursor.                     ;
;                                                                     ;
; Writes:       OLD_CURSOR_TYPE                                       ;
;---------------------------------------------------------------------;
set_cursor      proc    private uses ax cx si di bp
        call    GetCursor               ;Get the old cursor shape
        mov     old_cursor_type, ax     ;And save it
        mov     ch,dh                   ;Put start line into CH
        mov     cl,dl                   ;Put end line into CL
        mov     ah,1                    ;Set cursor type
        int     10h
        ret
set_cursor      endp
```

```
;----------------------------------------------------------------------;
; This procedure gets the start and end scan lines of the cursor.      ;
; If the cursor was not visible, it returns a visible cursor.  Call    ;
; GetCursor if you want the exact information on the old cursor.       ;
;                                                                      ;
; Returns:      AH      First scan line of cursor                      ;
;               AL      Last scan line of cursor                       ;
;----------------------------------------------------------------------;
GetVisibleCursor        proc
        call    GetCursor               ;Get cursor size into AX
        cmp     ah,0Fh                  ;Is the cursor off?
        jb      cursor_not_off          ;No, then we're all done
        mov     ax,607h                 ;Yes, set to 607 for CGA
        call    fix_cursor              ;Make sure CX contains a legal value
cursor_not_off:
        ret
GetVisibleCursor        endp


;----------------------------------------------------------------------;
; This procedure gets the start and end scan lines of the cursor.  You ;
; can then use SET_CURSOR to restore the cursor's shape.               ;
;                                                                      ;
; NOTE: There is a bug in the ROM BIOS.  Every time you set the mode,  ;
;       it stores 0607h into the cursor type, which is the wrong value ;
;       for a monochrome display.  So...this routine translates 0607h  ;
;       into 0B0Ch if the display is a monochrome display.             ;
;                                                                      ;
; Returns:      AH      First scan line of cursor.                     ;
;               AL      Last scan line of cursor.                      ;
;----------------------------------------------------------------------;
GetCursor       proc    uses bx cx dx si di bp
        xor     bh,bh                   ;Make sure page has reasonable value
        mov     ah,3                    ;Get current cursor type
        int     10h                     ;CX = cursor type
        mov     ax,cx                   ;Put cursor type into AX
        call    fix_cursor              ;Make sure CX contains a legal value
        ret
GetCursor       endp
```

```
;-----------------------------------------------------------------------;
; This procedure "fixes" the cursor start and end scan lines in case   ;
; the ROM BIOS gave us bogus information:                               ;
;                                                                       ;
; On entry:     AH      Starting scan line of the cursor                ;
;               AL      Ending scan line of the cursor                  ;
;                                                                       ;
;       67h             COMPAQ returns this incorrect result            ;
;       607h            ROM BIOS returns this for monochrome display    ;
;                                                                       ;
; Returns:      AX      Valid cursor size                               ;
;-----------------------------------------------------------------------;
        extrn   GetDisplayRows:procRef
fix_cursor      proc    private
        cmp     ax,67h                  ;Is this a bogus COMPAQ cursor?
        jne     not_bogus_compaq        ;No, then check for monochrome
        mov     ax,607h                 ;Make it a legal type
not_bogus_compaq:
        push    ax
        int     11h                     ;Get the equipment flag
        and     al,30h                  ;Leave only the display type
        cmp     al,30h                  ;Is this a monochrome display?
        pop     ax
        jne     cursor_type_valid       ;No, then accept cursor type

is_monochrome:
        push    ax
        call    GetDisplayRows          ;Get number of lines on screen
        cmp     ax,25                   ;Are we in 43-line mode?
        pop     ax
        ja      cursor_type_valid       ;Yes, keep cursor at 607h

        cmp     ax,607h                 ;Yes, is this a bogus cursor?
        jne     cursor_type_valid       ;No, then continue
        mov     AX,0B0Ch                ;Set to value for underscore cursor
cursor_type_valid:
        ret
fix_cursor      endp


;-----------------------------------------------------------------------;
; This procedure turns the cursor off.                                  ;
;                                                                       ;
; Uses:         SET_CURSOR                                              ;
;-----------------------------------------------------------------------;
```

```
CursorOff       proc    uses ax dx
        mov     dh,0Fh                  ;Set start line for cursor
        mov     dl,0                    ;Set end line for cursor
        call    set_cursor
        ret
CursorOff       endp


;------------------------------------------------------------------------;
; This procedure turns the cursor off.                                   ;
;                                                                        ;
; Uses:         set_cursor                                               ;
;                                                                        ;
; Note: This procedure preserves the registers so you can call it from  ;
;       assembly language.                                               ;
;------------------------------------------------------------------------;
CursorOn        proc    uses dx
        mov     dx,old_cursor_type      ;Restore old cursor type
        call    set_cursor              ;And set it.
        ret
CursorOn        endp


;------------------------------------------------------------------------;
; This procedure sets the cursor to a block cursor.  Since the color-   ;
; graphics and monochrome-display adapter use a different number of     ;
; lines for the character cell, this procedure has to check the display;
; type.                                                                  ;
;------------------------------------------------------------------------;
        extrn   GetDisplayType:procRef

CursorBlock     proc    uses ax bx
        xor     dh,dh                   ;Set start line for cursor to 0
        mov     dl,7                    ;Set for color-graphics adapter
        call    GetDisplayType
        cmp     al,3                    ;Is this a monochrome display?
        jne     block_is_graphics       ;No, then don't change CL
        mov     dl,13                   ;Set for monochrome display.
block_is_graphics:
        call    set_cursor              ;Change the cursor's shape
        ret
CursorBlock     endp
```

```
;-------------------------------------------------------------------;
; This procedure sets the cursor to a block cursor.  Since the color-  ;
; graphics and monochrome-display adapter use a different number of    ;
; lines for the character cell, this procedure has to check the display;
; type.                                                               ;
;-------------------------------------------------------------------;
        extrn   GetDisplayType:procRef

CursorUnderscore        proc    uses ax dx
        mov     dh,6                    ;Set start line for cursor to 0
        mov     dl,7                    ;Set for color-graphics adapter
        call    GetDisplayType
        cmp     al,3                    ;Is this a monochrome display?
        jne     underscore_is_graphics  ;No, then don't change CL
        mov     dh,11                   ;Set for monochrome display.
        mov     dl,12
underscore_is_graphics:
        call    set_cursor
        ret
CursorUnderscore        endp

        end

_
```

FASTWRIT.ASM

```
.MODEL  MEM_MODEL,C

INCLUDE sochalib.inc

;-------------------------------------------------------------------;
; Copyright © 1992 by John Socha                                     ;
;-------------------------------------------------------------------;

BELL    equ     7
TAB     equ     9
LF      equ     10
CR      equ     13
```

```
;----------------------------------------------------------------------;
; The procedures in this module are all designed for very fast writes  ;
; to the screen.  DOS is very slow, and so is the ROM BIOS.  So these  ;
; procedures poke characters directly, and very quickly, into memory.  ;
;                                                                      ;
; Because these procedures need special information, and it would be   ;
; ineffecient to constantly update local variables, there is one       ;
; procedure called INIT_FAST_DISPLAY_MODULE that initializes various   ;
; variables.  Call this procedure any time you change the display mode ;
; or screen.                                                           ;
;                                                                      ;
;       InitFastDisplayModule   Call this before calling other procs   ;
;       FastFlush               Flush unwritten chars to the screen    ;
;       FastWriteRawChar        Doesn't check for special characters   ;
;       FastWriteChar           Much faster version of WriteChar       ;
;       FastReadAttr            Returns attribute of char under cursor ;
;       FastWriteString         Much faster version of WriteString     ;
;       FastWriteUDecimal       Writes an unsigned-decimal number      ;
;       FastWriteNChars         Writes N copies of a character         ;
;       FastWriteSpaces         Fills an area with N spaces            ;
;       FastGotoXY              Version of GOTO_XY for FAST module     ;
;       FastSetCursor           Moves real cursor to screen_x, screen_y ;
;       FastGetXY               Version of GET_XY for FAST module      ;
;                                                                      ;
;       move_to_screen          Copies chars in the buffer to screen   ;
;       fast_write_raw_char     The low-level version                  ;
;       fast_write_char         The low-level version                  ;
;       calc_display_offset     (private) low-level function           ;
;       fast_goto_xy            (private) low-level function           ;
;                                                                      ;
; Public variables:                                                    ;
;       char_attribute          This is a public variable              ;
;       clear_attr              Used for new lines when scrolling      ;
;       display_lines           Number of lines on the screen          ;
;----------------------------------------------------------------------;


.DATA
        public  char_attribute, clear_attr, display_lines
char_attribute          db      7, 0    ;Attribute for characters
clear_attr              db      7, 0    ;Attribute for areas to clear
wait_retrace_flag       db      0, 0    ;1 if we should wait for horiz. retrace
topview_flag            db      0, 0    ;If we need to make update calls
```

```
display_base            dw      0B800h  ;For color graphics adapter
display_offset          dw      0       ;The display offset (used for page > 0)
display_page            db      0       ;The currently active page
display_lines           dw      25      ;Number of screen lines

screen_x                dw      0
screen_y                dw      0
screen_ptr              dw      0       ;Points to character under cursor

line_buffer             dw      80 DUP (0)
line_ptr                dw      line_buffer     ;Pointer to next char
line_count              dw      0       ;Number of characters to write
line_start              dw      0       ;Offset in screen memory of line start


max_lines       equ     50              ;Maximum number of screen lines
screen_offset           dw      0,      80,     160,    240,    320
                        dw      400,    480,    560,    640,    720
                        dw      800,    880,    960,    1040,   1120
                        dw      1200,   1280,   1360,   1440,   1520
                        dw      1600,   1680,   1760,   1840,   1920
                        dw      2000,   2080,   2160,   2240,   2320
                        dw      2400,   2480,   2560,   2640,   2720
                        dw      2800,   2880,   2960,   3040,   3120
                        dw      3200,   3280,   3360,   3440,   3520
                        dw      3600,   3680,   3760,   3840,   3920
                        dw      4000,   4080,   4160,   4240,   4320
                        dw      4400,   4480,   4560,   4640,   4720

.CODE

        extrn   GetDisplayType:procRef          ;In HARDWARE module
        extrn   EgaActive:procRef               ;In HARDWARE module
        extrn   GetDisplayRows:procRef          ;In HARDWARE module
        extrn   get_xy:procRef                  ;In CURSOR   module
        extrn   get_active_page:procRef         ;In CURSOR   module
;----------------------------------------------------------------------;
; Call this procedure at least once before you call *any* other        ;
; procedures.  This procedure initializes variables that other         ;
; procedures use.                                                      ;
;----------------------------------------------------------------------;
InitFastDisplayModule   proc
        push    ax
        push    dx
        mov     display_base,0B000h     ;Set up for monochrome card
        mov     display_lines,25        ;Set to 25 lines on the screen
        mov     wait_retrace_flag,0     ;Don't wait for retraces

        call    GetDisplayType          ;See which card we have
        cmp     al,3                    ;Do we have a monochrome card?
        je      is_monochrome           ;Yes, then we're done
        mov     display_base,0B800h     ;Set up for color graphics card
        call    EgaActive               ;See if we have EGA card (-1)
        or      ax,ax                   ;Is this EGA card?
        jnz     is_ega                  ;Yes, don't wait for retrace
        mov     wait_retrace_flag,1     ;Wait for retraces before writing chars
        jmp     Short finish_init

is_monochrome:
        ;-----------------------------------------;
        ; The display adapter is attached to an IBM    ;
        ; monochrome display, but it could be an EGA   ;
        ; card.                                        ;
        ;-----------------------------------------;
        call    EgaActive               ;See if EGA card (-1)
        or      ax,ax                   ;Is this an EGA/VGA card?
        Jz      finish_init             ;No, then we're done
                                        ;Yes, set display lines

is_ega:
        ;-----------------------------------------;
        ; This section of code gets the number of      ;
        ; screen lines for EGA and VGA monitors.       ;
        ;-----------------------------------------;
        call    GetDisplayRows          ;AX = Number of display lines
        cmp     al,max_lines            ;Too many lines?
        jbe     set_display_lines       ;No, then set
        mov     al,max_lines            ;Yes, set to max_lines
set_display_lines:
        mov     Byte Ptr display_lines,al

finish_init:
        ;-----------------------------------------;
        ; This section of code gets the current page   ;
        ; and calculates the offset.                   ;
        ;-----------------------------------------;
        push    cx
        call    get_active_page         ;Get current page number in AL
        mov     display_page,al         ;Save the display page
        mov     ah,al                   ;Put display page into AH
        xor     al,al                   ;AX = page * 256
        mov     cl,4                    ;Multiply AX by 16
        shl     ax,cl                   ;AX = page * 4096 (offset to page)
        mov     display_offset,ax       ;Save this page offset
        pop     cx



        ;-----------------------------------------------;
        ; Check TopView interface (also windows) for    ;
        ; address of the display buffer.                ;
        ;-----------------------------------------------;
        push    bx
        push    di
        push    es
        mov     bx,display_base         ;Get what we think is the display base
        mov     es,bx
        xor     di,di                   ;ES:DI - where we think display base is
        mov     ah,0FEh                 ;Make call to TopView
        int     10h                     ;Get Video Buffer
        mov     ax,es                   ;Get the new? display base
        cmp     ax,bx                   ;Are we running under TopView?
        je      not_in_topview          ;No, then we're done here
        mov     topview_flag,1          ;Yes we are, set flag
        mov     display_base,ax         ;Save new display base
        mov     display_offset,di       ;Save offset to the buffer
        mov     display_page,0          ;Use page 0 for TopView
        mov     wait_retrace_flag,0     ;No need to wait since TopView updates
not_in_topview:
        pop     es
        pop     di
        pop     bx

        ;-----------------------------------------------;
        ; Now read the default character attribute.     ;
        ;-----------------------------------------------;
        call    get_xy                  ;Get the current cursor position
        mov     dx,ax                   ;Set up for Fast_goto_xy call
        call    fast_goto_xy            ;Set the character offset
        call    FastReadAttr            ;Read the current attribute
        mov     char_attribute,al       ;Set character attribute to normal
        mov     clear_attr,al           ;And set for cleared regions
        pop     dx
        pop     ax
        ret
InitFastDisplayModule   endp
```

```
;------------------------------------------------------------------;
; This procedure writes any characters left in the buffer to the    ;
; screen.                                                            ;
;                                                                    ;
;       void FastFlush(void);                                        ;
;                                                                    ;
; Note: this subroutine preserves all the registers so you can call it ;
;       from assembly language.                                      ;
;------------------------------------------------------------------;
FastFlush       proc uses ax cx si di es
        mov     cx,line_count           ;Number of characters to display
        jcxz    done_fast_flush         ;If no chars, we're all done

        lea     ax,line_buffer          ;Pointer to off-screen buffer
        mov     line_ptr,ax             ;Reset line_ptr to start of buffer
        mov     si,ax                   ;DS:SI points to line buffer

        mov     ax,line_start           ;Where to start drawing the line
        mov     di,ax
        mov     ax,display_base         ;Segment for the screen buffer
        mov     es,ax                   ;ES:DI points to where we'll draw

        call    move_to_screen          ;Copy the buffer to the screen
        mov     ax,line_start           ;Get the start of the line
        mov     cx,line_count           ;Get num of char/attr pairs in buffer
        shl     cx,1                    ;Convert to number of bytes
        add     ax,cx                   ;Point to first char after line
        mov     screen_ptr,ax           ;Point to next location
        mov     line_start,ax
        mov     line_count,0            ;Reset the line count

done_fast_flush:
        ret
FastFlush       endp


        extrn   HideMouse:procRef
        extrn   UnhideMouse:procRef
;------------------------------------------------------------------;
; This procedure moves a section of memory directly to screen memory. ;
; It waits for horizontal retrace periods, where needed, to avoid    ;
; snow.                                                              ;
;                                                                    ;
; On entry:     DS:SI           Pointer to data to move to screen    ;
;               DI              Offset in screen of first char       ;
;               CX              Number of words to transfer to screen  ;
;                                                                      ;
; Destroys:     CX, SI, DI                                             ;
;----------------------------------------------------------------------;
move_to_screen  proc    private uses ax es
        call    HideMouse               ;Hide cursor while we write
        jcxz    done_move_to            ;CX == 0, we're all done

        mov     ax,display_base         ;Seg register of frame buffer
        mov     es,ax                   ;Set segment to start of frame buffer

        push    cx                      ;Save the character count
        push    di                      ;Save start of screen area
        cld                             ;Set for increment
        test    wait_retrace_flag,1     ;Should we wait for horizontal retrace?
        jnz     move_to_wait            ;Yes.

rep     movsw                           ;No, just move to the screen
        jmp     Short finish_move_to    ;We're all done

move_to_wait:                           ;We need to wait for hoizontal retrace.
        push    bx
        push    dx

        mov     dx,03DAh                ;Status register on color card
move_to_loop:
        lodsw                           ;Get character from off-screen buffer
        mov     bx,ax                   ;Save character in BX register
to_still_in_retrace:
        in      al,dx                   ;Get status byte
        test    al,1                    ;In horizontal retrace?
        jnz     to_still_in_retrace     ;Yes, wait until it's done
        cli                             ;Turn interrupts off for now
to_wait_for_retrace:
        in      al,dx                   ;Get status byte
        test    al,1                    ;In horizontal retrace?
        jz      to_wait_for_retrace     ;Not yet, wait
        mov     ax,bx                   ;Recover character and attribute
        stosw                           ;Save it to the screen
        sti                             ;Turn interrupts back on
        loop    move_to_loop            ;Copy another word.

        pop     dx
        pop     bx
```

```
finish_move_to:
        pop     di                      ;Get back start of screen area
        pop     cx                      ;Get back number of characters

        test    topview_flag,1          ;Are we in TopView?
        jz      done_move_to            ;No, we're all done
        mov     ah,0FFh                 ;Tell TopView to update screen
        int     10h

done_move_to:
        call    UnhideMouse             ;Show the mouse again
        ret
move_to_screen  endp


;---------------------------------------------------------------------;
; This procedure provides the C-language interface to the function    ;
; below.                                                              ;
;---------------------------------------------------------------------;
FastWriteRawChar        proc    char:Word
        mov     dl, Byte Ptr char       ; Get the character to display
        call    fast_write_raw_char     ; Display the character
        ret
FastWriteRawChar        endp


;---------------------------------------------------------------------;
; This procedure is like WRITE_CHAR, except that it writes characters ;
; directly to the screen and it doesn't attach special meaning to any ;
; characters, so CR, LF, etc. all appear on the screen as characters  ;
;                                                                     ;
; NOTE: This procedure will set SCREEN_X to 80 if you try to write off ;
;       the right side of the screen.  This way you can tell when you ;
;       tried to write off the screen.                                ;
;                                                                     ;
; Note: This procedure is used by other assembly-language procedures  ;
;       in this file.                                                 ;
;                                                                     ;
; On entry:     DL      Character you want to print.                  ;
;                                                                     ;
; Reads:        CHAR_ATTRIBUTE                                        ;
;---------------------------------------------------------------------;
fast_write_raw_char     proc    private uses ax di es
        push    ds
        pop     es                      ;Set ES so it points to data segment
        cmp     screen_x,79             ;Are we still on screen?
        ja      done_write_raw          ;No, don't write this char
        mov     di,line_ptr             ;Get pointer to the next char
        mov     ah,char_attribute       ;Get current character attribute
        mov     al,dl                   ;Get the character to write
        stosw                           ;Save this character
        inc     screen_x                ;Move cursor to next column
        inc     line_count              ;Number of characters in buffer
        mov     line_ptr,di             ;Save new character pointer

done_write_raw:
        ret
fast_write_raw_char     endp



;-------------------------------------------------------------------;
; This procedure provides the C-language interface to fast_write_char: ;
;                                                                   ;
;       void FastWriteChar(int c);                                  ;
;-------------------------------------------------------------------;
FastWriteChar   proc    char:Word
        mov     dl, Byte Ptr char       ; Get the character to display
        call    fast_write_char         ; Display the character
        ret
FastWriteChar   endp


;-------------------------------------------------------------------;
; This is a version of WRITE_CHAR that writes characters directly to   ;
; the screen.  It uses attribute 7 as the default attribute.        ;
;                                                                   ;
; The following characters all need special treatment:              ;
;       7       Bell                                                ;
;       9       Tab                                                 ;
;       10      Line Feed                                           ;
;       13      Carriage Return                                     ;
;                                                                   ;
; On entry:     DL      Character you want to print.                ;
;-------------------------------------------------------------------;
fast_write_char         proc    private
        test    dl,0F0h                 ;Special character?
        jz      special                 ;Yes, then handle them
```

```
not_special:
        call    fast_write_raw_char     ;No, the write character to screen
        ret

special:
        cmp     dl,CR                   ;Is it a carriage return?
        je      do_cr                   ;Yes, then take care of CR
        cmp     dl,LF                   ;Is it a line feed?
        je      do_lf                   ;Yes, then take care of LF
        cmp     dl,TAB                  ;Is it a tab?
        je      do_tab                  ;Yes, then take care of tab
        cmp     dl,BELL                 ;Is it a bell?
        je      do_bell                 ;Yes, then sound the bell
        jmp     Short not_special       ;Not a special charcter
                                        ;(It's below all the DO routines)

do_cr:                                  ;Move the cursor to the left side
        push    dx
        mov     dh,Byte Ptr screen_y
        xor     dl,dl                   ;Move to start of this line
        call    fast_goto_xy
        pop     dx
        ret

do_lf:
        push    ax
        mov     ax,display_lines        ;Get number of lines on screen
        dec     ax                      ;AX == last line on screen
        cmp     screen_y,ax             ;At or past bottom of screen?
        pop     ax
        jae     passed_bottom           ;Yes, then scroll the screen
        push    dx
        mov     dh,Byte Ptr screen_y
        inc     dh                      ;Move to the next line
        mov     dl,Byte Ptr screen_x    ;But same column
        call    fast_goto_xy            ;Move the cursor
        pop     dx
        ret

passed_bottom:
        push    ax
        push    bx
        push    cx
        push    dx
        push    si
        push    di
        push    bp
        mov     ax,0601h                ;Scroll up by one line
        mov     bh,char_attribute       ;Use char. attribute in scrolled lines
        mov     cx,0                    ;Upper left corner at (0,0)
        mov     dh,Byte Ptr display_lines ;Lower right at (display_lines, 79)
        dec     dh                      ;Convert to last screen line
        mov     dl,4Fh
        int     10h                     ; ***Replace with CALL to SCROLL_WINDOW
        pop     bp
        pop     di
        pop     si
        pop     dx
        pop     cx
        pop     bx
        pop     ax
        ret

do_bell:                                ;Ring the bell.
        ret

do_tab:                                 ;Expand the tab into spaces.
        push    cx
        push    dx
        mov     dl,' '                  ;Write out N spaces
        mov     cx,screen_x
        and     cx,7                    ;CX := SCREEN_X MOD 8
        neg     cx
        add     cx,8                    ;CX := 8 - (SCREEN_X MOD 8)
tab_loop:
        call    fast_write_raw_char
        loop    tab_loop
        pop     dx
        pop     cx
        ret
fast_write_char         endp


;-------------------------------------------------------------------;
; This procedure reads the attribute of the character under the cursor ;
;                                                                   ;
; Returns:                                                          ;
;       AX      Attribute of character under cursor                 ;
;-------------------------------------------------------------------;
```

```
FastReadAttr    proc    uses dx di es
        mov     ax,display_base
        mov     es,ax
        mov     di,screen_ptr
        inc     di                      ;Point to the attribute

;-------------------------------------------;
; This section of code waits for the horizontal  ;
; retrace period before writing characters so    ;
; you won't see snow on an IBM color-graphics    ;
; adapter.  Since the horizontal retrace period  ;
; is only about 2 microseconds, there's only     ;
; time to read one character/attribute.          ;
;-------------------------------------------;
        test    wait_retrace_flag,1     ;Should we wait for horizontal retrace?
        jz      read_dont_wait          ;Nope

        mov     dx,03DAh                ;Status register on color card
read_still_in_retrace:
        in      al,dx                   ;Get status byte
        test    al,1                    ;In horizontal retrace?
        jnz     read_still_in_retrace   ;Yes, wait until it's done
        cli                             ;Turn interrupts off for now
read_wait_for_retrace:
        in      al,dx                   ;Get status byte
        test    al,1                    ;In horizontal retrace?
        jz      read_wait_for_retrace   ;Not yet, wait
read_dont_wait:
        mov     al,es:[di]              ;Get the attribute
        sti                             ;Turn interrupts back on again
        xor     ah,ah                   ;Return attribute in a word

        ret
FastReadAttr    endp


;-----------------------------------------------------------------------;
; This procedure writes an ASCIIZ string to the screen.  It uses the    ;
; FAST_WRITE_CHAR so it's very fast, and only a few characters are      ;
; treated specially.                                                    ;
;                                                                       ;
; On entry:     DS:DX   Address of ASCIIZ string.                       ;
; Uses:         FAST_WRITE_CHAR                                         ;
;-----------------------------------------------------------------------;
FastWriteString proc    uses ax dx si, string:Ptr Byte
        lodES   si,string               ;Get the address of the string
```

```
write_string_loop:
        mov     al,refES [si]           ;Get one character
        inc     si                      ;Point to next character
        or      al,al                   ;Is the end of the string?
        jz      end_of_string           ;Yes, then stop writing
        mov     dl,al                   ;No, then print the character
        call    fast_write_char
        jmp     Short write_string_loop
end_of_string:
        ret
FastWriteString endp


;---------------------------------------------------------------------;
; This procedure writes an unsigned decimal number.                   ;
;                                                                     ;
; On entry:     DX      Number you want to write in decimal           ;
; Uses:         fast_write_char                                       ;
;---------------------------------------------------------------------;
FastWriteUDecimal       proc    num:Word
        mov     ax, num                 ;Put number in AX, where we want it
        mov     bx,10                   ;We divide by 10 to pick off digits
        xor     cx,cx                   ;Start with 0 digits on the stack
not_zero:
        xor     dx,dx                   ;Set upper word of 32-bit num to 0
        div     bx                      ;(N mod 10) --> DX, (N div 10) -- AX
        push    dx                      ;Save this digit on the stack
        inc     cx                      ;Keep track of num. of digits on stack
        or      ax,ax                   ;N == 0 yet?
        jnz     not_zero                ;No, put more digits on stack
write_digit_loop:
        pop     dx                      ;Get most recent digit
        add     dl, '0'                 ;Convert to '0'..'9'
        call    fast_write_char         ;Display this character
        loop    write_digit_loop        ;Until no more digits on stack
        ret
FastWriteUDecimal       endp


;---------------------------------------------------------------------;
; This procedure writes N characters, starting at the cursor position. ;
;                                                                     ;
;       void FastWriteNChars(int char, int count);                    ;
;                                                                     ;
; Calls:        fast_write_raw_char                                   ;
;---------------------------------------------------------------------;
```

```
FastWriteNChars proc    uses cx, char:Word, count:Word
        mov     dl, Byte Ptr char       ;Get the character to display
        mov     cx, count               ;Number of chars to show
        jcxz    done_fast_write_chars   ;Count == 0, nothing to write
chars_loop:
        call    fast_write_raw_char     ;Write out one character
        loop    chars_loop              ;Keep writing until CX == 0
done_fast_write_chars:
        ret
FastWriteNChars endp


;-----------------------------------------------------------------------;
; This procedure writes N spaces, starting at the cursor position.     ;
;                                                                       ;
; On entry:     CX      Number of spaces we should write                ;
; Calls:        FAST_WRITE_N_CHARS                                      ;
;-----------------------------------------------------------------------;
FastWriteSpaces proc    count:Word
        INVOKE  FastWriteNChars, ' ', count
        ret
FastWriteSpaces endp


;-----------------------------------------------------------------------;
; This procedure calculates the offset into the screen buffer of a     ;
; given character on the screen.  (Note: this procedure handles screen ;
; pages and TopView correctly.)                                         ;
;                                                                       ;
; On entry:     DH      Row     Y                                       ;
;               DL      Column  X                                       ;
; Returns:      Offset, in bytes, from the start of the screen segment ;
;               for the character at (X, Y).                            ;
;-----------------------------------------------------------------------;
calc_display_offset     proc    private
        push    bx
        mov     bl,dh                   ;Get the line number
        xor     bh,bh                   ;Convert to a word
        shl     bx,1                    ;Multiply line by 2 for lookup table
        mov     ax,screen_offset[bx]    ;Get character offset for this line

        mov     bl,dl                   ;Get the X position
        xor     bh,bh                   ;Convert to a word
        add     ax,bx                   ;Character offset of cursor
        shl     ax,1                    ;Byte offset of character/attribute
        add     ax,display_offset       ;Add in offset from non-zero page
        pop     bx
        ret
calc_display_offset     endp


;------------------------------------------------------------------;
; This procedure provides the C interface to fast_goto_xy:         ;
;                                                                  ;
;       void FastGotoXY(x, y);                                     ;
;------------------------------------------------------------------;
FastGotoXY      proc    x:Word, y:Word
        mov     dh, Byte Ptr y
        mov     dl, Byte Ptr x
        call    fast_goto_xy
        ret
FastGotoXY      endp

        extrn   goto_xy:procRef         ;In CURSOR module
;------------------------------------------------------------------;
; This is a version of GOTO_XY that works with the other fast screen   ;
; procedures.  This procedure moves the cursor to another position on  ;
; the screen.  It doesn't do any error checking in case you want to do ;
; something off the screen.  But be careful!                       ;
;                                                                  ;
; On entry:     DH      Row(Y)                                     ;
;               DL      Column (X)                                 ;
; Uses:         GOTO_XY                                            ;
;------------------------------------------------------------------;
fast_goto_xy    proc    private uses ax bx dx
        call    FastFlush               ;Flush any unwritten characters
        mov     bl,Byte Ptr display_lines ;Temporary storage for Num lines
        dec     bl                      ;Convert to last line on screen
        cmp     dh,bl                   ;Are we off the screen?
        jbe     do_goto_xy              ;No, then move the cursor
        cmp     dh,bl                   ;Are we off the bottom?
        mov     dh,bl                   ; (Set DH to the last line)
        jg      do_goto_xy              ;Yes, then set to last line
        mov     dh,0                    ;No, we're off the top, set to line 0
do_goto_xy:
        call    goto_xy                 ;Move the real cursor
        mov     Byte Ptr screen_x,dl    ;Save coordinates of cursor
        mov     Byte Ptr screen_y,dh
        call    calc_display_offset     ;Get byte offset into AX
        mov     screen_ptr,ax           ;Save this offset
        mov     line_start,ax           ;Remember where next line starts
        ret
fast_goto_xy    endp


;-----------------------------------------------------------------------;
; This procedure will move the real cursor to the position of the      ;
; fast cursor, which is just a pair of numbers stored in memory.       ;
;                                                                       ;
; Note: This procedure preserves the registers so you can call it from ;
;       assembly language.                                              ;
;-----------------------------------------------------------------------;
FastSetCursor   proc    uses dx
        mov     dl,Byte Ptr screen_x
        mov     dh,Byte Ptr screen_y
        call    goto_xy                 ;Move the real cursor
        ret
FastSetCursor   endp


;-----------------------------------------------------------------------;
; This is a version of FAST_GET_XY that works with the fast screen     ;
; module procedures.  It's the reverse of FAST_GOTO_XY.  Instead of    ;
; moving the cursor to a new position, it returns the current cursor;
; position.                                                             ;
;                                                                       ;
;       void FastGetXY(int *x, int *y);                                 ;
;                                                                       ;
; Returns:      x       Column                                          ;
;               y       Row                                             ;
;-----------------------------------------------------------------------;
FastGetXY       proc    x:Ptr Word, y:Ptr Word
        lodES   bx, x                   ; Get pointer to x into es:[bx]
        mov     ax, screen_x            ; Get current x position
        mov     refES [bx], ax          ; Save into C's x variable

        lodES   bx, y                   ; Get pointer to y into es:[bx]
        mov     ax, screen_y            ; Get current y position
        mov     refES [bx], ax          ; Save into C's y variable
        ret
FastGetXY       endp


if 0
```

```
        PUBLIC  FAST_CLEAR_TO_EOL
        EXTRN   CLEAR_WINDOW:NEAR
;-----------------------------------------------------------------------;
; This procedure clears the line from the current cursor position to   ;
; the end of the line.                                                  ;
;-----------------------------------------------------------------------;
        PUBLIC  _FAST_CLEAR_TO_EOL
_FAST_CLEAR_TO_EOL      LABEL   NEAR
FAST_CLEAR_TO_EOL       PROC    NEAR
        PUSH    AX
        PUSH    BX
        MOV     BL,BYTE PTR SCREEN_X    ;Get position of cursor
        MOV     BH,BYTE PTR SCREEN_Y
        MOV     AL,79                   ;Last column of the line
        CMP     BL,AL                   ;Was last char on screen?
        JA      DONT_CLEAR_TO_EOL       ;No, then don't try to clear
        MOV     AH,BH                   ;Stay on same line
        CALL    CLEAR_WINDOW
DONT_CLEAR_TO_EOL:
        POP     BX
        POP     AX
        RET
FAST_CLEAR_TO_EOL       ENDP

endif

        end
```

HARDWARE.ASM

```
.MODEL  MEM_MODEL,C

INCLUDE sochalib.inc

;-----------------------------------------------------------------------;
; JS -- Copyright © 1992 by John Socha                                  ;
;                                                                       ;
; This file contains a number of procedures that check or set various  ;
; things having to do with the hardware.                                ;
;                                                                       ;
;       GetDisplayType          Tells you what type of display active   ;
;       is_ega (private)        Checks to see if this is EGA adapter    ;
;       EgaActive               Checks if EGA is the active card        ;
;       GetDisplayRows          Returns number of rows on screen        ;
;-----------------------------------------------------------------------;
rom_seg         segment at 40h
        org     4Ah
crt_cols        DW      ?               ;Number of columns on the screen

        org     87h
ega_info        DB      ?               ;Info byte for EGA monitor
rom_seg         ends


.CODE



;-----------------------------------------------------------------------;
; This procedure determines whether the display type is a monochrome    ;
; or color-graphics card.                                               ;
;                                                                       ;
; Returns:      AX      0       No display cards                        ;
;                       1       Color card in 40x25 mode                ;
;                       2       Color card in 80x25 mode                ;
;                       3       If monochrome card                      ;
; Destroys:     AH                                                      ;
;-----------------------------------------------------------------------;
GetDisplayType  proc    uses cx
        int     11h                     ;Get equipment flags
        mov     cl,4
        shr     ax,cl                   ;Right justify display flags
        and     ax,3                    ;Keep just the display flags
        ret
GetDisplayType  endp


;-----------------------------------------------------------------------;
; This procedure checks to see if the display adapter is an EGA card.   ;
; This test is from the IBM Personal Computer Seminar Proceedings       ;
; Volume 2, Number 11 from November 1984                                ;
;                                                                       ;
; Returns:      ZR      If this is an EGA card                          ;
;               NZ      Otherwise                                       ;
;-----------------------------------------------------------------------;
```

```
is_ega  proc    private uses ax bx cx si di bp
        mov     ax,1200h                ;Make an EGA-specific call
        mov     bl,10h                  ;For EGA information
        mov     bh,0FFh                 ;Load BH with invalid info
        mov     cl,0Fh                  ;Load CL with reserved switch settings
        int     10h                     ;Make call to EGA ROM BIOS

        cmp     cl,0Ch                  ;Is switch setting in valid range?
        jae     not_ega                 ;No, this is not an EGA card
        cmp     bh,1                    ;Is this a valid flag?
        ja      not_ega                 ;No, this is not an EGA card
        cmp     bl,3                    ;Is memory value within range?
        ja      not_ega                 ;No, this is not an EGA card
        xor     ax,ax                   ;Set the zero flag for EGA card

done_is_ega:
        ret

not_ega:
        xor     ax,ax
        inc     ax                      ;Clear the zero flag
        jmp     done_is_ega
is_ega  endp


;-------------------------------------------------------------------------;
; This procedure tests to see if an EGA monitor is the active monitor. ;
;                                                                       ;
; Returns:      0       EGA card is not the active card                 ;
;              -1       An EGA or VGA card *is* active                  ;
;-------------------------------------------------------------------------;
EgaActive       proc    uses es
        call    is_ega                  ;See if have an EGA card installed
        jnz     ega_not_active          ;No, report EGA not active

        mov     ax,rom_seg              ;Point to ROM BIOS data area
        mov     es,ax
        test    es:ega_info,8           ;Is this card active?
        jnz     ega_not_active          ;No
        xor     ax,ax                   ;Yes, return -1 since EGA card found
        not     ax
        jmp     done_ega_active
```

```
ega_not_active:
        xor     ax,ax                   ;Return 0 for no EGA card active

done_ega_active:
        ret
EgaActive       endp


;-----------------------------------------------------------------------;
; This procedure returns the number of rows on the screen:             ;
;                                                                       ;
; Returns:      AX      25              All non-EGA/VGA displays        ;
;               > 25            EGA/VGA displays with more lines        ;
;-----------------------------------------------------------------------;
GetDisplayRows  proc uses bx si di bp es
        call    EgaActive               ;Return 0 if no EGA card
        or      ax,ax                   ;Is this an EGA card?
        mov     al,25                   ;Return 25 for non-EGA displays
        jz      done_get_rows           ;No, return 25 lines

        mov     ax,1130h                ;Ask for EGA information
        xor     bh,bh                   ;Return the current information
        int     10h                     ;DL = Number of rows - 1
        inc     dl                      ;DL = Number of rows
        mov     al,dl                   ;Returns result in AX
done_get_rows:
        xor     ah,ah                   ;Set upper byte to 0
        ret
GetDisplayRows  endp


        end
_
```

KBD_IO.ASM

```
.MODEL  MEM_MODEL,C

INCLUDE sochalib.inc
```

```
;-------------------------------------------------------------------------;
; Copyright © 1992 by John Socha                                          ;
;-------------------------------------------------------------------------;
;-------------------------------------------------------------------------;
; This file contains the following procedures:                            ;
;                                                                         ;
; ReadKey()             Reads a character from the ROM BIOS and returns   ;
;                       either an ASCII code, or 0x1?? where ?? is the    ;
;                       scan code of the non-ASCII key.                   ;
; scan_to_extended      Converts to extended ASCII code (1xxh)            ;
; ReadRawKey()          Returns scan code in high byte and character      ;
;                       code in the low byte.                             ;
; ClearKbdBuffer()      Clears the keyboard buffer                        ;
; KeyWaiting()          Returns code of next character if the buffer is   ;
;                       not empty, and 0 if it is empty.                  ;
; ShiftKeys()           Returns the current state of the shift keys.      ;
;-------------------------------------------------------------------------;

.CODE

        public ReadKey
;-------------------------------------------------------------------------;
; This procedure uses the ROM BIOS routines to read a single character.   ;
; It returns the ASCII code for keys that generate an ASCII character.    ;
; But if you push one of the key-pad or function keys, this procedure     ;
; will return the scan code in the lower byte and set the upper byte      ;
; to 1.                                                                   ;
;                                                                         ;
; Returns:      ASCII code for ASCII characters                           ;
;               1xx (hex) where xx is the scan code of the special key    ;
;-------------------------------------------------------------------------;
ReadKey proc
        xor     ah,ah                   ;Ask for a character from BIOS
        int     16h
        call    scan_to_extended        ;Convert to 1xx for special keys
        cld                             ;Must be cleared for MSC
        ret
ReadKey endp
```

```
;---------------------------------------------------------------------;
; This procedure converts a scan-code/ASCII-code pair into an extended  ;
; character.  If the lower byte is 0, this procedure puts the scan      ;
; code into the lower byte and sets the upper byte to 1.  Otherwise, it ;
; sets the upper byte to 0.                                             ;
;                                                                       ;
; On entry:     AH      Scan code for the key pushed                    ;
;               AL      ASCII code, or 0 for special keys               ;
;                                                                       ;
; Returns:      ASCII code for ASCII characters                         ;
;               1xx (hex) where xx is the scan code of the special key  ;
;                                                                       ;
; Note:         This procedure now works also for the grey keys on the  ;
;               extended keyboard (which have E0h in the lower byte     ;
;               rather than 00h).                                       ;
;---------------------------------------------------------------------;
scan_to_extended        proc    private
        or      al,al                   ;Is this a special key?
        je      special_key             ;Yes, then handle it
        cmp     al,0E0h                 ;Is this a special key?
        je      special_key             ;Yes, then handle it
        xor     ah,ah                   ;No, return the ASCII code
        ret

special_key:
        xchg    al,ah                   ;Put the scan code into AL
        mov     ah,1                    ;And set AH = 1
        ret
scan_to_extended        endp


;---------------------------------------------------------------------;
; This procedure reads one character (or one special function key) from ;
; the keyboard and returns the ASCII code, or scan code.                ;
;                                                                       ;
; This procedure uses the ROM BIOS call so it can read both the         ;
; character code AND the scan code.  The DOS call returns only the      ;
; ASCII code.                                                           ;
;                                                                       ;
; Returns:      AH      The scan code of the key you pushed.  This is   ;
;                       useful for special keys that don't have ASCII   ;
;                       codes, like the function and cursor keys.       ;
;               AL      The ASCII code for the key you pushed.          ;
;                       0 for all special keys.                         ;
;---------------------------------------------------------------------;
```

```
ReadRawKey      proc
        mov     ah,0                    ;Ask for a character
        int     16h
        cld                             ;Must be cleared for MSC
        ret
ReadRawKey      endp


;-----------------------------------------------------------------------;
; This procedure clears the keyboard buffer of any characters that may ;
; still be around.                                                      ;
;-----------------------------------------------------------------------;
ClearKbdBuffer  proc
        push    dx
        mov     ax,0C06h                ;Clear input buffer
        mov     dl,0FFh                 ;Check status of the return
        int     21h
        pop     dx
        cld                             ;Must be cleared for MSC
        ret
ClearKbdBuffer  endp


;-----------------------------------------------------------------------;
; This procedure checks the keyboard buffer to see if there are any    ;
; characters waiting to be read from the ROM BIOS:                      ;
;                                                                       ;
; Returns:      Character code of next character waiting.               ;
;               -1 if there are no characters waiting.                  ;
;                                                                       ;
; NOTE: The ROM BIOS returns a scan code and character code of 0 after ;
;       you've hit ^break, so it is possible to detect a ^break by     ;
;       checking to see if you read a 0 from the keyboard.              ;
;                                                                       ;
; NOTE: This procedure issues an INT 28h before checking the keyboard  ;
;       status to allow background processes more frequent access to   ;
;       system resources.                                               ;
;-----------------------------------------------------------------------;
KeyWaiting      proc
        int     28h                     ;Allow background processes a slice

        mov     ah,1                    ;Check for waiting character
        int     16h
        jnz     done_kbd_hit            ;Return the character code
        xor     ax,ax                   ;Return -1 for no character
        not     ax
done_kbd_hit:
        cld                             ;Must be clear for MSC compiler
        ret
KeyWaiting      endp


;--------------------------------------------------------------------;
; This procedure returns the current state of the shift keys:        ;
;                                                                    ;
; INS_STATE     80h     Insert state is turned on                    ;
; CAPS_STATE    40h     Caps lock key is on                          ;
; NUM_STATE     20h     Num lock is turned on                        ;
; SCROLL_STATE  10h     Scroll lock is on                            ;
; ALT_SHIFT     08h     The alt key is down                          ;
; CTL_SHIFT     04h     The control key is down                      ;
; LEFT_SHIFT    02h     The left-hand shift key is down              ;
; RIGHT_SHIFT   01h     The right-hand shift key is down             ;
;--------------------------------------------------------------------;
ShiftKeys       proc
        mov     ah,2
        int     16h                     ;Get the current flags
        xor     ah,ah                   ;Set upper byte to zero
        cld                             ;Must be cleared for MSC
        ret
ShiftKeys       endp


        end

_
```

MOUSE.ASM

```
.MODEL  MEM_MODEL,C

INCLUDE sochalib.inc
```

```
;-----------------------------------------------------------------------;
; Copyright © 1992 by John Socha                                         ;
;                                                                       ;
; This file contains some of the mouse interface routines.              ;
;                                                                       ;
; InitMouse();          Do a full initlize on mouse hardware and sw.    ;
; InitMouseSw();        Initialize the mouse software, and return       ;
;                               -1      Mouse installed                 ;
;                               0       No mouse installed              ;
; HideMouser();         Removes the mouse cursor from the screen        ;
; UnhideMouse();        Makes the mouse cursor visible again            ;
;                                                                       ;
; Get_mouse_position(&x, &y);           Returns character coords        ;
; Set_mouse_position(x, y);             Returns character coords        ;
; Mouse_buttons(&x, &y);                Which and where buttons pushed ;
;-----------------------------------------------------------------------;

.DATA
        public  mouse_installed, swap_buttons, mouse_visible
mouse_installed         DB      0               ;TRUE if a mouse is installed
swap_buttons            DW      0               ;TRUE to swap mouse buttons
mouse_visible           DB      0               ;Visible if > 0
                        DB      0               ;Make it a word

mouse_cursor            DW      0               ;Storage for mouse cursor char


.CODE


;-----------------------------------------------------------------------;
; This procedure tests to see if the mouse driver is installed, or if  ;
; some driver is installed on INT 33h.                                  ;
;                                                                       ;
; Returns:      AX       0      There is no mouse installed             ;
;                       -1      There is a driver on INT 33h            ;
;-----------------------------------------------------------------------;
check_int_33    proc    private uses bx es
        mov     ax,3533h                ;Get the current INT 33h vector
        int     21h
        mov     ax,es                   ;Put the segment into AX
        or      ax,ax                   ;Is anything installed?
        jz      no_int_33               ;No, then return 0
        or      bx,bx                   ;Is anything installed?
        jz      no_int_33               ;No, then return 0
        mov     ax,0FFFFh               ;Return -1, there is INT 33h vector
        jmp     Short done_check_int_33

no_int_33:
        xor     ax,ax
done_check_int_33:
        ret
check_int_33    endp


;-----------------------------------------------------------------------;
; This procedure initializes the mouse by calling function 0, which is ;
; a full reset of the mouse software AND hardware.  In some cases this ;
; can take several seconds (a serial mouse or an IBM mouse), which may ;
; not be acceptable.  Init_mouse() can be much faster in such cases.   ;
; Here are the rules on which of these two procedures to call:         ;
;                                                                       ;
;       Once                                                            ;
;               Init_mouse() must be called at least to initialize     ;
;               the mouse hardware the first time since not all mouse  ;
;               drivers initialize the mouse proplerly when you do     ;
;               a software mouse reset.                                 ;
;                                                                       ;
;       After Init_mouse()                                              ;
;               From then on you can call Init_mouse_sw() for a fast   ;
;               reset.  The Commander calls Init_mouse() once, then    ;
;               Init_mouse_sw() each time you return to the Commander  ;
;               after running a DOS command.                            ;
;-----------------------------------------------------------------------;
InitMouse       proc
        mov     bx,0                    ;Ask for a full reset
        call    do_init_mouse           ;Initialize the mouse
        ret
InitMouse       endp


;-----------------------------------------------------------------------;
; This procedure initializes the mouse.  It tries to do a software     ;
; reset first.  Only if the software reset doesn't work does it try to ;
; do a hardware reset.  In some cases (such as a serial mouse or the   ;
; IBM mouse) a hardware reset can take as much as 5 seconds, which is  ;
; not an acceptable delay when returning from the Commander.           ;
;-----------------------------------------------------------------------;
```

```
InitMouseSw     proc
        mov     bx,1                    ;Ask for a software reset
        call    do_init_mouse           ;Initialize the mouse
        ret
InitMouseSw             endp


;-------------------------------------------------------------------;
; This procedure resets the mouse software, and checks to see if the   ;
; mouse and mouse software are installed.  It returns the following:    ;
;                                                                   ;
; On entry:     BX      0       Do a hardware reset                 ;
;                       1       Do a soft reset                     ;
;                                                                   ;
; Returns:      AX      -1      The mouse and software are installed   ;
;                       0       There is no mouse, driver not installed;
;-------------------------------------------------------------------;
        extrn   EgaActive:procRef
do_init_mouse   proc    private uses bx cx dx es bp
        call    check_int_33            ;See INT 33 has vector before we call
        or      ax,ax                   ;Is anything installed?
        jz      no_mouse                ;No, report that there's no mouse

        or      bx,bx                   ;Yes, Is this a full hardware reset?
        jz      do_hardware_reset       ;Yes, do a full hardware reset

do_reset:                               ;No, try to reset the mouse software
        mov     ax,33                   ;Request a software reset
        int     33h                     ;Call the mouse routine
        cmp     ax,33                   ;Did mouse ignore this function?
        je      do_hardware_reset       ;Yes, do a hardware reset
        cmp     ax,-1                   ;Is the mouse installed?
        jne     do_hardware_reset       ;No, then do a hardware reset
        cmp     bx,2                    ;Is the mouse installed?
        je      finish_init             ;Yes, then finish initializing

do_hardware_reset:
        mov     ax,0                    ;Ask for the mouse reset function
        int     33h                     ;Call the mouse routine
        or      ax,ax                   ;Is the mouse installed?
        jz      no_mouse                ;No, report that mouse is not installed

finish_init:
        mov     ax,10                   ;Set the text cursor mask and type
        xor     bx,bx                   ;Select the software text cursor
        mov     cx,0FFFFh               ;Set the mask to all ones
        mov     dx,7700h                ;And set cursor to inverting cursor
        int     33h                     ;Returns -1 if if the mouse installed
        mov     mouse_visible,0         ;The mouse is now hidden

        call    EgaActive               ;See if an EGA card is active
        or      ax,ax                   ;Is EGA active?
        jz      done_init_is_mouse      ;No, then we're all done
        mov     ax,1130h                ;Get EGA information
        xor     bh,bh                   ;Ask for the default information
        int     10h                     ;DL = #rows-1, CX = char height, pixels
        inc     dl                      ;DL = # rows
        cmp     dl,25                   ;Is this normal 25 line mode?
        je      done_init_is_mouse      ;Yes, don't change mouse settings
        mov     al,dl                   ;Put number of rows into AX
        mov     cl,8                    ;Microsoft assumes chars 8 pixels high
        mul     cl                      ;AX = screen height, in pixels
        dec     ax                      ;AX = screen height - 1
        mov     dx,ax                   ;DX = screen height - 1
        mov     ax,8                    ;Set max and min vertical positions
        mov     cx,0
        int     33h

done_init_is_mouse:
        xor     ax,ax                   ;Return -1 to report mouse installed
        not     ax

done_init_mouse:
        mov     mouse_installed,al      ;Save the mouse installed state
        ret

no_mouse:
        xor     ax,ax                   ;The mouse is not installed
        jmp     done_init_mouse
do_init_mouse   endp


;-----------------------------------------------------------------------;
; This procedure puts the mouse cursor back on the screen.  I unhide    ;
; the mouse only when Mouse_visible is 0 when you make this call.  In   ;
; other words, only when you cross the visiblity threshold.             ;
;                                                                       ;
; You can nest calls to Hide and Unhide_mouse_cursor.                   ;
;-----------------------------------------------------------------------;
```

```
UnhideMouse     proc
        test    mouse_installed,0FFh    ;Is the mouse installed?
        jz      done_unhide             ;No, don't do anything
        inc     mouse_visible
        cmp     mouse_visible,1         ;Did mouse just become visible?
        jne     done_unhide             ;No, then don't unhide again
        push    ax                      ;Yes, make the INT 33 call
        mov     ax,1                    ;Ask for the show-cursor function
        int     33h
        pop     ax
done_unhide:
        ret
UnhideMouse     endp


;-----------------------------------------------------------------------;
; This procedure removes the mouse cursor from the screen.             ;
;-----------------------------------------------------------------------;
HideMouse       proc
        test    mouse_installed,0FFh    ;Is the mouse installed?
        jz      done_hide               ;No, don't do anything
        dec     mouse_visible           ;Decrement the count
        cmp     mouse_visible,0         ;Did mouse just become invisible?
        jne     done_hide               ;No, then don't hide mouse again
        push    ax
        mov     ax,2                    ;Ask for the hide-cursor function
        int     33h
        pop     ax
done_hide:
        ret
HideMouse       endp


;-----------------------------------------------------------------------;
; This procedure returns the current mouse position in character       ;
; coordinates.                                                          ;
;                                                                       ;
;       GetMousePosition(int *x, int *y);                               ;
;-----------------------------------------------------------------------;
GetMousePosition        proc    x:Ptr Word, y:Ptr Word
        test    mouse_installed,0FFh    ;Is the mouse installed?
        jz      done_get_mouse          ;No, then don't do anything
        mov     ax,3                    ;Ask for get position function
        int     33h                     ;Ask for the mouse position
        lodES   bx,x                    ;Get address of X
        shr     cx,1                    ;Divide result by 8 to get char. coord.
        shr     cx,1
        shr     cx,1
        mov     refES [bx],cx           ;Save X coordinate
        shr     dx,1                    ;Divide by 8 to get char. coord.
        shr     dx,1
        shr     dx,1
        lodES   bx,y                    ;Get address of Y
        mov     refES [bx],dx           ;Save Y coordinate
done_get_mouse:
        ret
GetMousePosition        endp


;-------------------------------------------------------------------;
; This procedure sets the mouse position on the screen in character   ;
; coordinates.                                                      ;
;                                                                   ;
;       SetMousePosition(int x, int y);                             ;
;-------------------------------------------------------------------;
SetMousePosition        proc    x:Word, y:Word
        test    mouse_installed,0FFh    ;Is the mouse installed?
        jz      done_set_mouse          ;No, then don't do anything
        mov     ax,4                    ;Ask for set mouse position function
        mov     cx,x                    ;Get the new X coordinate
        shl     cx,1                    ;Multiply by 8 to get pixel coords
        shl     cx,1
        shl     cx,1
        mov     dx,y                    ;Get the new Y coordinate
        shl     dx,1                    ;Multiply by 8 to get pixel coords
        shl     dx,1
        shl     dx,1
        int     33h
done_set_mouse:
        ret
SetMousePosition        endp


;-------------------------------------------------------------------;
; This procedure checks the status of the mouse buttons:            ;
;                                                                   ;
;       int MouseButtons(int *x, int *y);                           ;
;                                                                   ;
; It returns the following information:                             ;
;                                                                   ;
;               0       Both buttons up                                 ;
;               1       Left button down                                ;
;               2       Right button down                               ;
;               3       Both buttons down                               ;
;                                                                       ;
; The X and Y coordinates returned are character coordinates to make   ;
; things easier since my stuff is currently character based.            ;
;                                                                       ;
; NOTE: This procedure treats the middle button on three-button mice   ;
;       as the same as both buttons.                                    ;
;-----------------------------------------------------------------------;
        extrn   FastFlush:procRef
MouseButtons    proc    x:Ptr Word, y:Ptr Word
        xor     ax,ax                   ;Return 0 if mouse not installed
        test    mouse_installed,0FFh    ;Is the mouse installed?
        jz      done_mouse_buttons      ;No, then return 0
        call    FastFlush               ;Flush unwritten chars to screen
        mov     ax,3                    ;Ask for button status
        int     33h
        mov     ax,bx                   ;Put button status into AX
        cmp     ax,3                    ;Is center or both buttons down?
        jb      middle_not_down         ;No, then continue.
        mov     ax,3                    ;Yes, set to both buttons
        jmp     Short finish_buttons    ;Now calculate coordinates
middle_not_down:
        or      ax,ax                   ;Are both buttons up?
        jz      finish_buttons          ;Yes, then calculate coordinates
        cmp     swap_buttons,0          ;Should we swap mouse buttons?
        jz      finish_buttons          ;No, calculate coordinates
        xor     ax,3                    ;Swap the mouse buttons
finish_buttons:
        lodES   bx,x                    ;Get address for X
        shr     cx,1                    ;Divide by 8 to return char. coords.
        shr     cx,1
        shr     cx,1
        mov     refES [bx],cx           ;Save X coordinate of mouse
        lodES   bx,y                    ;Get address for Y
        shr     dx,1                    ;Divide by 8 to return char. coords.
        shr     dx,1
        shr     dx,1
        mov     refES [bx],dx           ;Save Y coordinate of mouse
done_mouse_buttons:
        ret
MouseButtons    endp

        end
```

A P P E N D I X

Miscellaneous Tables

The eight reference tables in this appendix provide listings of values used for setting character and keyboard codes, screen display colors, and interrupts.

D

Topics Covered

Table D-1 ASCII Character Codes

Decimal	Hex	Graphic Character	Decimal	Hex	Graphic Character
000	00	null	028	1C	FS
001	01	☺	029	1D	GS
002	02	☻	030	1E	RS
003	03	♥	031	1F	US
004	04	♦	032	20	SP
005	05	♣	033	21	!
006	06	♠	034	22	"
007	07	●	035	23	#
008	08	■	036	24	$
009	09	○	037	25	%
010	0A	■	038	26	&
011	0B	♂	039	27	'
012	0C	♀	040	28	(
013	0D	♪	041	29	)
014	0E	♫	042	2A	*
015	0F	☼	043	2B	+
016	10	►	044	2C	,
017	11	◄	045	2D	-
018	12	↕	046	2E	.
019	13	‼	047	2F	/
020	14	¶	048	30	0
021	15	§	049	31	1
022	16	–	050	32	2
023	17	↨	051	33	3
024	18	↑	052	34	4
025	19	↓	053	35	5
026	1A	→	054	36	6
027	1B	←	055	37	7

Decimal	Hex	Graphic Character	Decimal	Hex	Graphic Character
056	38	8	086	56	V
057	39	9	087	57	W
058	3A	:	088	58	X
059	3B	;	089	59	Y
060	3C	<	090	5A	Z
061	3D	=	091	5B	[
062	3E	>	092	5C	\
063	3F	?	093	5D	]
064	40	@	094	5E	^
065	41	A	095	5F	–
066	42	B	096	60	`
067	43	C	097	61	a
068	44	D	098	62	b
069	45	E	099	63	c
070	46	F	100	64	d
071	47	G	101	65	e
072	48	H	102	66	f
073	49	I	103	67	g
074	4A	J	104	68	h
075	4B	K	105	69	i
076	4C	L	106	6A	j
077	4D	M	107	6B	k
078	4E	N	108	6C	l
079	4F	O	109	6D	m
080	50	P	110	6E	n
081	51	Q	111	6F	o
082	52	R	112	70	p
083	53	S	113	71	q
084	54	T	114	72	r
085	55	U	115	73	s

Decimal	Hex	Graphic Character	Decimal	Hex	Graphic Character
116	74	t	146	92	Æ
117	75	u	147	93	ô
118	76	v	148	94	ö
119	77	w	149	95	ò
120	78	x	150	96	û
121	79	y	151	97	ù
122	7A	z	152	98	ÿ
123	7B	{	153	99	Ö
124	7C	¦	154	9A	Ü
125	7D	}	155	9B	¢
126	7E	~	156	9C	£
127	7F	DEL	157	9D	¥
128	80	Ç	158	9E	₧
129	81	ü	159	9F	ƒ
130	82	é	160	A0	á
131	83	â	161	A1	í
132	84	ä	162	A2	ó
133	85	à	163	A3	ú
134	86	å	164	A4	ñ
135	87	ç	165	A5	Ñ
136	88	ê	166	A6	ª
137	89	ë	167	A7	º
138	8A	è	168	A8	¿
139	8B	ï	169	A9	⌐
140	8C	î	170	AA	¬
141	8D	ì	171	AB	½
142	8E	Ä	172	AC	¼
143	8F	Å	173	AD	¡
144	90	É	174	AE	«
145	91	æ	175	AF	»

Decimal	Hex	Graphic Character	Decimal	Hex	Graphic Character
176	B0	░	206	CE	╬
177	B1	▒	207	CF	╧
178	B2	▓	208	D0	╨
179	B3	│	209	D1	╤
180	B4	┤	210	D2	╥
181	B5	╡	211	D3	╙
182	B6	╢	212	D4	╘
183	B7	╖	213	D5	╒
184	B8	╕	214	D6	╓
185	B9	╣	215	D7	╫
186	BA	║	216	D8	╪
187	BB	╗	217	D9	┘
188	BC	╝	218	DA	┌
189	BD	╜	219	DB	█
190	BE	╛	220	DC	▄
191	BF	┐	221	DD	▌
192	C0	└	222	DE	▐
193	C1	┴	223	DF	▀
194	C2	┬	224	E0	α
195	C3	├	225	E1	β
196	C4	─	226	E2	Γ
197	C5	┼	227	E3	π
198	C6	╞	228	E4	Σ
199	C7	╟	229	E5	σ
200	C8	╚	230	E6	µ
201	C9	╔	231	E7	τ
202	CA	╩	232	E8	Φ
203	CB	╦	233	E9	θ
204	CC	╠	234	EA	Ω
205	CD	═	235	EB	δ

Decimal	Hex	Graphic Character	Decimal	Hex	Graphic Character
236	EC	∞	246	F6	÷
237	ED	ø	247	F7	≈
238	EE	∈	248	F8	°
239	EF	∩	249	F9	•
240	F0	≡	250	FA	·
241	F1	±	251	FB	√
242	F2	≥	252	FC	η
243	F3	≤	253	FD	²
244	F4	⌠	254	FE	■
245	F5	⌡	255	FF	

Table D-2 Color Codes

0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Violet
6	Brown
7	White

Attribute = background color * 16 + foreground color

Add 8 to the foreground color for the bright versions, or add 8 to the background color to turn on blinking.

Table D-3 Extended Keyboard Codes

Many of the keys on the keyboard (such as the function keys) return a two-character code when you read the keys through DOS: a decimal 0 followed by a scan code. The following table shows the scan codes for all the keys that have no equivalent ASCII code.

15	Shift Tab
16-25	Alt keys for Q, W, E, R, T, Y, U, I, O, P
30-38	Alt keys for A, S, D, F, G, H, J, K, L
44-50	Alt keys for Z, X, C, V, B, N, M
59-68	F1 through F10
71	Home
72	Cursor Up
73	PgUp
75	Cursor Left
77	Cursor Right
79	End
80	Cursor Down
81	PgDn
82	Ins
83	Del
84-93	Shift F1 through F10
94-103	Control F1 through F10
104-113	Alt F1 through F10
114	Control PrtSc
115	Control Left Cursor
116	Control Right Cursor
117	Control End
118	Control PgDn
119	Control Home
120-131	Control Alt for 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, =
132	Control PgUp

Table D-4 Table of Addressing Modes

Addressing Mode	*Format of Address*	*Segment Register Used*
Register	register (such as AX)	None
Immediate	data (such as 12345)	None

continues

Addressing Mode	*Format of Address*	*Segment Register Used*
Memory Addressing Modes		
Register Indirect	[BX]	DS
	[BP]	SS
	[DI]	DS
	[SI]	DS
Base Relative*	label[BX]	DS
	label[BP]	SS
Direct Indexed*	label[DI]	DS
	label[SI]	DS
Base Indexed*	label[BX+SI]	DS
	label[BX+DI]	DS
	label [BP+SI]	SS
	label[BP+DI]	SS
String Commands:		Read from DS:SI
(MOVSW, LODSB, *and so on)*		Write to ES:DI

* Label[...] can be replaced by [disp+...], where *disp* is a displacement. Thus, we could write [10+BX] and the address would be 10 + BX.

Table D-5 INT 10h Functions

(AH)=0 **Set the display mode.** The AL registers contains the mode number.

	Text Modes
(AL)=0	40 by 25, black and white mode
(AL)=1	40 by 25, color
(AL)=2	80 by 25, black and white
(AL)=3	80 by 25, color
(AL)=7	80 by 25, monochrome display adapter

		Graphics Mode
	(AL)=4	320 by 200, color
	(AL)=5	320 by 200, black and white
	(AL)=6	640 by 200, black and white
(AH)=1	**Set the cursor size.**	
	(CH)	Starting scan line of the cursor. The top line is 0 on both the monochrome and color graphics displays, while the bottom line is 7 for the color graphics adapter and 13 for the monochrome adapter. Valid range: 0 to 31.
	(CL)	Last scan line of the cursor.

The power-on setting for the color graphics adapter is CH=6 and CL=7. For the monochrome display: CH=11 and CL=12.

(AH)=2	**Set the cursor position.**	
	(DH,DL)	Row, column of new cursor position; the upper left corner is (0,0).
	(BH)	Page number. This is the number of the display page. The color-graphics adapter has room for several display pages, but most programs use page 0.
(AH)=3	**Read the cursor position.**	
	(BH)	Page number
On exit	(DH,DL)	Row, column of cursor
	(CH,CL)	Cursor size
(AH)=4	**Read light pen position** (see Tech. Ref. Man.).	
(AH)=5	**Select active display page.**	
	(AL)	New page number (from 0 to 7 for modes 0 and 1; from 0 to 3 for modes 2 and 3)

(AH)=6 **Scroll up.**

(AL)	Number of lines to blank at the bottom of the window. Normal scrolling blanks one line. Set to zero to blank entire window.
(CH,CL)	Row, column of upper left corner of window
(DH,DL)	Row, column of lower right corner of window
(BH)	Display attribute to use for blank lines

(AH)=7 **Scroll down.**

Same as scroll up (function 6), but lines are left blank at the top of the window instead of the bottom

(AH)=8 **Read attribute and character under the cursor.**

(BH)	Display page (text modes only)
(AL)	Character read
(AH)	Attribute of character read (text modes only)

(AH)=9 **Write attribute and character under the cursor.**

(BH)	Display page (text modes only)
(CX)	Number of times to write character and attribute on screen
(AL)	Character to write
(BL)	Attribute to write

(AH)=10 **Write character under cursor** (with normal attribute).

(BH)	Display page
(CX)	Number of times to write character
(AL)	Character to write

(AH)=11 to 13 **Various graphics functions.** (See Tech. Ref. Man. for the details)

(AH)=14 **Write teletype.** Write one character to the screen and move the cursor to the next position.

(AL)	Character to write
(BL)	Color of character (graphics mode only)
(BH)	Display page (text mode)

(AH)=15 **Return current video state.**

(AL)	Display mode currently set
(AH)	Number of characters per line
(BH)	Active display pages

Table D-6 INT 16h Functions

This table contains the INT 16h functions used in this book to read characters from the keyboard.

(AH)=0 **Keyboard read.** This function waits for you to type a character
on the keyboard. It returns the ASCII code in AL and the scan code in AH. For extended keys, AL will be set to 0. See Table D-2 for a list of scan codes for such keys.

(AL)	ASCII code of the key you press (0 for special keys).
(AH)	Scan code for the key you pressed.

(AH)=1 **Keyboard status.** This function checks to see if there are any keys waiting to be read.

ZF	0 if a character is waiting, 1 if there are no characters waiting.
(AL)	ASCII code of character waiting to be read.
(AH)	Scan code of character waiting to be read.

(AH)=2	**Shift status.** This function returns a byte with the state of the various shift keys:
	(AL) Status of the shift keys:

7 6 5 4 3 2 1	
1	Insert on
. 1	Caps Lock on
. . 1	Num Lock on
. . . 1 . . .	Scroll Lock on
. . . . 1 . .	Alt key down
. . . . . 1 .	Left shift down
. 1	Right shift down

Table D-7 INT 21h Functions

This table contains the INT 21h functions used in this book. For a more complete list, you should buy the IBM *DOS Technical Reference* manual.

(AH)=1 **Keyboard input.** This function waits for you to type a character on the keyboard. It echoes the character to the screen, and returns the ASCII code in the AL register. For extended keyboard codes, this function returns two characters: an ASCII 0 followed by the scan code (see Table D-2).

(AL)	Character read from the keyboard.

(AH)=2 **Display output.** Displays one character on the screen. Several characters have special meaning to this function:

7	Beep: Send a one-second tone to the speaker.
8	Backspace: move the cursor left one character position.
9	Tab: Move to the next tab stop. Tab stops are set to every 8 characters.

	0Ah	Line feed: Move to the next line.
	0Dh	Carriage return: Move to the start of the current line.
	(DL)	Character to display on the screen.
(AH)=8	**Keyboard input without echo.** Reads a character from the keyboard, but doesn't display the character on the screen.	
	(AL)	Character read from keyboard.
(AH)=9	**Display string.** Displays the string pointed to by the DS:DX pair of registers. You must mark the end of the string with the $ character.	
	DS:DX	Points to the string to display.
(AH)=0Ah	**Read string.** Reads a string from the keyboard. See Chapter 23 for more details.	
(AH)=25h	**Set interrupt vector.** Sets an interrupt vector to point to a new routine.	
	(AL)	Interrupt number.
	DS:DX	Address of the new interrupt handler.
(AH)=35h	**Get interrupt vector.** Gets the address of the interrupt service routine for the interrupt number given in AL.	
	(AL)	Interrupt number.
	ES:BX	Address of the interrupt handler.
(AH)=4Ch	**Exit to DOS.** Returns to DOS, like INT 20h, but it works for both .COM and .EXE programs. The INT 20h function works only for .COM programs.	
	(AL)	Return code. Normally set to 0, but you can set it to any other number and use the DOS batch commands IF and ERRORLEVEL to detect errors.

Table D-8 Sector Read/Write Functions

The following two interrupts are DOS calls for reading and writing disk sectors.

INT 25h—Read Disk Sector

On entry:

(AL)	Drive number (0=A, 1=B, and so on)
(CX)	Number of sectors to read at one time
(DX)	Number of the first sector to read (the first sector is 0)
DS:BX	Transfer address: where to write the sectors read

INT 26h—Write Disk Sector

On entry:

(AL)	Drive number (0=A, 1=B, and so on)
(CX)	Number of sectors to write at one time
(DX)	Number of the first sector to write (the first sector is 0)
DS:BX	Transfer address: start of the data we want to write to the disk.

Information Returned by INT 25h, INT 26h

Both INT 25h and INT 26h return the following information in the AX register. They also leave the flags on the stack, so you'll want to use a POP or POPF to remove this word from the stack (see Chapter 15 for an example).

Returns:

Carry Flag	Set if there was an error, in which case the error information will be in AX.
(AL)	DOS error code
(AH)	Contains one of the following:

80h	The drive did not respond
40h	The Seek operation failed
08h	Bad CRC when we read the disk
04h	Could not find the sector we asked for
03h	Tried to write to a write-protected disk
02h	Some other error

Destroys

AX, BX, CX, DX, SI, DI, BP

Index

[illegible]

[illegible]

LIMITED WARRANTY REGISTRATION CARD

[illegible]

DISK REPLACEMENT ORDER FORM

In the event that the disk bound in to this book is defective, Prentice Hall Computer Publishing will send you a replacement disk free of charge.

Please fill out the information below and return this card to the address listed below with your original disk. Please print clearly.

BOOK TITLE ______________________ ISBN ____________

NAME ______________________ PHONE ____________

COMPANY ______________________ TITLE ____________

ADDRESS ______________________

CITY ______________ STATE ________ ZIP ____________

Prentice Hall Computer Publishing, 11711 North College Avenue, Carmel IN 46032.
ATTN: Customer Service Department.

LIMITED WARRANTY REGISTRATION CARD

In order to preserve your rights as provided in the limited warranty, this card must be on file with PHCP within thirty days of purchase.

Please fill in the information requested:

BOOK TITLE ______________________ ISBN ____________

NAME ______________________ PHONE NUMBER (   ) ____________

ADDRESS ______________________

CITY ______________ STATE ________ ZIP ____________

COMPUTER BRAND & MODEL ____________ DOS VERSION ____________ MEMORY ________ K

Where did you purchase this product?

DEALER NAME? ______________________ PHONE NUMBER (   ) ____________

ADDRESS ______________________

CITY ______________ STATE ________ ZIP ____________

PURCHASE DATE ______________ PURCHASE PRICE ____________

How did you learn about this product? (Check as many as applicable.)

STORE DISPLAY ______ SALESPERSON ______ MAGAZINE ARTICLE ______ ADVERTISEMENT ______

OTHER (Please explain) ______________________

How long have you owned or used this computer?

LESS THAN 30 DAYS ______ LESS THAN 6 MONTHS ______ 6 MONTHS TO A YEAR ______ OVER 1 YEAR ______

What is your primary use for the computer?

BUSINESS ______ PERSONAL ______ EDUCATION ______ OTHER (Please explain) ______________

Where is your computer located?

HOME ______ OFFICE ______ SCHOOL ______ OTHER (Please explain) ______________

PUT
FIRST
CLASS
STAMP
HERE

Prentice Hall Computer Publishing
11711 N. College Avenue
Carmel, IN 46032

Attn: **Order Department**

FORMATTED
for IBM PC AT's
and other 1.2 MB systems

Assembly Language/Programming/IBM
All Levels

A Building Block Tutorial for Utilizing the Tremendous Speed and Power of Assembly Language!

Peter Norton and John Socha, creators of The Norton Utilities and The Norton Commander, show you how to exploit the full potential of assembly language. Start by learning the instructions of the 80x86 microprocessor and then quickly move on to more advanced topics that are invaluable when building industrial-strength programs. With *Assembly Language for the PC*, you get:

- An extensive review of hexadecimal arithmetic, software interrupts, and assembly procedures and subroutines
- Step-by-step instructions for building and modifying a large multi-file assembly language program - you'll see step-wise refinements and modular design at work
- Insider tips and tricks on using memory segments, using assembly code for Windows programs, and protected-mode programming for the 386/486 systems
- A guide to linking C/C++ and assembly language code and to including assembly language code in C/C++ files using in-line assembly

By the time you've finished reading this book, you'll know how to write assembly language programs and you'll know how to use assembly language in your C/C++ programs. Along the way, you'll learn many techniques professional programmers use to simplify their programs, to dramatically increase their programming speed and performance, and to write more readable and reliable programs.

The companion disk contains both the tutorial files as well as an advanced version DSKPATCH, a powerful disk editor; DISKLITE, a complete RAM resident program; and WINLIB, a library that checks pointers in Windows programs to see whether they are valid. You also get more than 30 assembly routines for C/C++ programming that provide mouse support, writing to your screen, some keyboard input routines, and much, much more.

Microsoft Macro Assembler 6 recommended. Most programs will also work with MASM 5.1 and Borland's Turbo Assembler.

ISBN 1-56686-016-4

$39.95 U.S.
$49.95 CAN
£37.60 (incl. VAT) Net UK

Brady Books/Prentice Hall
Computer Publishing