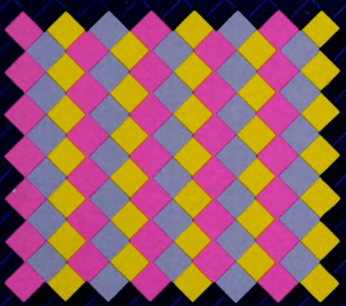
PLUME
WAITE

The Waite Group

ASSEMBLY LANGUAGE PRIMER for the IBM® PC & XT



Features Color Graphics and DOS Function Calls

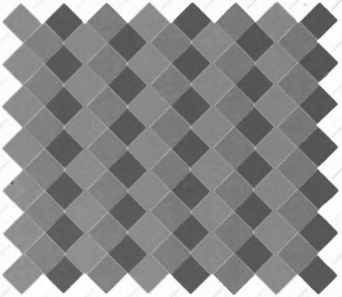
- Unique, Easy-to-Learn Approach
- Graphics and Sound Programming
- Includes DEBUG, Disk Access, and File Handles
- Covers Novice to Advanced Levels

by Robert Lafore

PLUME
WAITE

The Waite Group

ASSEMBLY LANGUAGE PRIMER for the IBM® PC & XT



Features Color Graphics and DOS Function Calls

- Unique, Easy-to-Learn Approach
- Graphics and Sound Programming
- Includes DEBUG, Disk Access, and File Handles
- Covers Novice to Advanced Levels

by Robert Lafore

ASSEMBLY LANGUAGE PRIMER

for the IBM® PC & XT

by

Robert Lafore

A Plume/Waite Book
New American Library
New York and Scarborough, Ontario

Several trademarks and/or service marks appear in this book. The companies listed below are the owners of the trademarks and/or service marks following their names.

International Business Machines Corporation: IBM, IBM PC, IBM Personal Computer, IBM PC XT, PC-DOS
Microsoft: MS-DOS, MBASIC
Digital Research: CP/M, CP/M-86
MicroPro International Corporation: WordStar
Apple Computer Inc.: Apple
Intel Corporation: Intel
SoftTech Microsystems: UCSD p-System
Epson Corporation: Epson
Atari Inc.: ATARI
Lotus: Lotus 1-2-3
Information Unlimited Software: EasyWriter
ATT Corporation: Bell Laboratories, Unix
ComputerLand
KayPro
Osborne
Xerox Corporation

LIBRARY OF CONGRESS CATALOGING IN PUBLICATION DATA

Lafore, Robert (Robert W.)
Assembly language primer for the IBM PC & XT.

"A Plume/Waite book."
Includes index.
1. IBM Personal Computer—Programming. 2. IBM Personal Computer XT—Programming. 3. Assembler language (Computer program language) I. Title.
QA76.8.I2594L34 1984 001.64'2 84-3302
ISBN: 0-452-25711-5

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN and NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, N.Y. 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L 1M8

Book and cover design by Dan Cooper
Typography by Walker Graphics

First Printing, May, 1984

5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

Contents

**This book is dedicated to the Munchkins,
without whose patient support the Emerald City
would still be only a myth.**

ACKNOWLEDGMENTS

The author would like to thank Mitchell Waite, who edited the manuscript of this book, suggested many important improvements, and provided moral support; John Angermeyer, whose technical expertise eliminated a variety of errors; and Janet Hunter, whose painstaking attention to detail was essential to the finished work.

Introduction

The purpose of this book is to teach you how to write programs in assembly language. Why would you want to study a computer language which has acquired the reputation of being somehow mysterious and difficult to learn?

Assembly language is always the fastest and most powerful language available for a given computer. It is essential in programs where pure speed of operation is important, such as graphics, sorting, and sustained number-crunching. It is also the only language that can make use of *all* of a particular machine's hardware features. With higher-level languages, such as BASIC or Pascal, the programmer is always insulated from the computer by the language itself — you can only do what the writers of the language decided you should be able to do. Inevitably, then, you can not tap the full power of the computer.

For these reasons, many types of programs, such as operating systems, compilers, word processors, and graphics programs, are almost always written in assembly language. So, if you want to do this sort of programming, you need to know assembly language.

But assembly language is not just practical, it is also a fascinating and rewarding field of study. It is so closely tied to the physical reality of the computer that it does not suffer from the somewhat arbitrary quality of higher-level languages. Everything you do in assembly language is the result of the way the computer operates, not the way the designers of a higher-level language decided to do things for the sake of ease and convenience.

We can think of higher-level languages as being like stodgy luxury sedans: they're comfortable and easy to use, but the steering is imprecise, the suspension insulates you from the feel of the road, and if you try to push them too fast they slide into the ditch.

Assembly language, on the other hand, is the sports car of computer languages. In a sports car you're close to the road. The steering, brakes and gears are light and precise, and the car is built for speed and efficiency. It may not be quite as comfortable as a sedan, but it's fast, and more importantly, it's fun to drive.

Assembly language is fun in the same way: it's fast, it's efficient, and it gives you the satisfaction of having complete control over a powerful and finely-tuned machine.

Is Assembly Language Really So Hard to Learn?

Unfortunately, assembly language has developed the reputation of being difficult to learn. Many people — even those who had no trouble learning a higher-level language such as BASIC — think that assembly language is somehow beyond them. This belief is fostered by many books on assembly language, which, strange as it may seem, appear to be written with the assumption that the reader already knows all about the subject the book is attempting to teach! For instance, many assembly-language books start off by listing and describing *all* of the scores of machine instructions. This is a good bit like giving a student in a first-year French class a dictionary, and telling him that as soon as he's memorized it he can go on to the next lesson! There must be an easier way.

We believe that assembly language, in spite of its reputation, is actually not too much harder to learn than any other computer language, provided that it is presented gradually and easily, so that the reader does not feel overwhelmed at the beginning. It's this sort of easy, step-by-step presentation that we have attempted to achieve in this book. For this reason we've avoided "clever" programming; that is, shortcuts which increase the speed or compactness of the program at the expense of clarity. Once a program has been written in an obvious way, it can always be modified to make it faster or smaller. Like poetry, very compact programs can be beautiful once you understand them, but require far more time to understand than a more obvious, less compact routine.

Why Is This Book Unusual?

Assembly Language Primer for the IBM PC and XT is unusual — and we believe superior to other books on the market — in several respects. First, it not only teaches assembly language, it *teaches it in the context of a particular computer*: the IBM PC. As we'll see, this provides significant advantages over books that try to cover *all* the computers which use a particular microprocessor chip.

Second, this book makes use of the built-in *DOS function calls*, which vastly simplify programming and can make even short programs powerful.

Third, to make things easy for the complete novice, this book makes extensive use of IBM's *DEBUG utility*, which provides a far simpler and less threatening introduction to assembly-language programming than more conventional approaches that plunge you immediately into the

complexities of a full MACRO Assembler program.

Finally, this book uses the graphics and sound capabilities of the IBM PC. This makes the learning experience more interesting. Plus, by making use of these features, you can ensure that the programs you write will be fun to use. If you decide to market your programs, graphics and sound will make them more popular and profitable.

Why Learn Assembly Language on the IBM PC?

If you are interested in writing programs for commercial use, the answer to the question posed above must be obvious: the IBM enjoys unprecedented sales growth. If you write a popular program for the IBM PC, you are guaranteed one of the largest markets in the personal computer field. There are other reasons, however, why the IBM PC is an especially appropriate computer on which to learn assembly language.

First, both the hardware and the software on the IBM PC are top quality. They are solid and reliable. You don't have to worry — as you do on some machines — that a hardware failure will suddenly cause a system crash and destroy a file you've spent hours creating, or that a mysterious bug in the assembler will prevent your program from assembling correctly, even though it is correctly written.

Second, if you want to be in the forefront of what's happening in computers, it's important to learn about the new 16-bit technology. The *internal* characteristics of the 8088 are 16-bit. This makes the IBM PC an ideal "stepping stone" to using an 8086 16-bit system.

Finally, the IBM PC Disk Operating System (PC-DOS) is far more powerful and versatile than earlier microcomputer operating systems. By writing programs under this operating system you ensure not only that your programs can make use of an extensive number of powerful DOS functions, but that you are learning how a sophisticated, state-of-the-art system operates. You also benefit from the fact that PC-DOS is very similar to (and is in fact derived from) another operating system, MS-DOS. By writing programs that run under PC-DOS you can (if you follow a few simple rules) ensure that the same programs will run under MS-DOS. MS-DOS is used on many non-IBM computers, so if you are interested in marketing your product, you will have a program you can sell to IBM PC owners and to owners of a host of other computers as well.

Who This Book Is For

This book is primarily intended for the person who has no previous

experience in assembly language programming: the rank beginner. However, it will benefit the programmer who knows assembly language for a different microprocessor, such as the 8085, Z-80, or 6502, and who wants to learn how the 8088 family of chips work

The Rank Beginner

If you have never written in assembly language, and have only a vague idea what it's all about, then this book is for you. We start at the very beginning, without inflated expectations about your knowledge of the subject.

Although the reputation that assembly language has for being difficult to learn is largely undeserved, many people still find it a bit less obvious than the simpler higher-level languages such as BASIC and Pascal. For this reason, we recommend that you have some experience with a higher-level language before you read this book. Although it is possible to learn assembly language as a first computer language, it's probably easier to cut your teeth on BASIC.

Once you know a little about a higher-level language, you'll not only understand in general what computer languages are supposed to do, but you will also have picked up the jargon and some of the ideas that are necessary for a real understanding of computers.

The Experienced Assembly Language Programmer

Although this book is oriented toward the beginner, you will still find it valuable if you are an experienced assembly-language programmer who is not yet familiar with the 8088 microprocessor and its implementation in the IBM PC. You may whiz through the book faster than the beginner, but even the initial chapters will be of interest, since it's here that you will learn how to use DEBUG — an essential tool — and various other useful skills.

In fact, if you are used to 8-bit microprocessors, you will find the 16-bit 8088 to be, in many ways, a whole new ball game. The use of memory segmentation, the extensive instruction set, the implementation of graphics and sound, the string-handling instructions, and the multiple addressing modes all require thorough examination, which this book provides.

The Equipment You Need to Use This Book

In this section we're going to discuss the equipment, both hardware and software, you need to best profit from this book.

Hardware

This is very much a "hands-on" book. Although you can gain a general understanding of assembly language by reading it without a computer at your disposal, you will be far better off if you have the computer on your desk before you start to read. As with other computer languages (and non-computer languages) it's only through practice that real mastery is achieved.

So we'll assume that you have access to an IBM Personal Computer, either a model with one or two floppy diskette drives, or the newer model with the fixed disk: the PC XT. You definitely can *not* use the cassette-based version of the PC, since the assembler program, various other software, and the entire DOS function approach used in this book, all require the disk operating system. Very few IBM PCs are sold in the cassette configuration, but if yours is one of them, rush out today and buy a set of floppy disk drives. If you're serious about computers, you won't regret it.

Memory Size and the Assembler

How big a memory do you need to create assembly-language programs? That depends which assembler you want to use. When you buy the standard IBM MACRO-Assembler, you actually get two assemblers in the same package: MASM and ASM. MASM stands for "Macro-ASseMbler," and is the full-scale assembler with all the bells and whistles. If you use this program you'll need a minimum of 96K, and you'll find that 128K is more useful.

ASM, which is sometimes called the "Small Assembler," is a more modest version of MASM. It leaves out some of MASM's more advanced features, such as MACROs and conditional assembly, and in consequence requires considerably less memory space. ASM will run in a 64K system if you are using PC-DOS version 1.00 or 1.10, but again you will probably be happier with more memory — 96K or 128K — especially if you plan to write large programs. However, if you are using DOS version 2.00, then you will need a minimum of 96K, with 128K being preferable.

Since this book does not describe MACROs and conditional assembly, there's no problem using ASM. In fact, ASM even has some advantages: since it's smaller, it loads faster and takes up less space on your disk. Thus we use ASM throughout the book (although you can use MASM if you want, and if you have enough memory).

So the answer to how much memory you need is: an absolute minimum of 64K, provided you are using DOS version 1.00 or 1.10, and ASM. However, we recommend that you upgrade to 128K if you can.

Display Monitors

You can use this book with any of the display options available on the PC: either the monochrome monitor used with a monochrome adapter board, an RGB (red, blue, green) color monitor, a non-IBM black and white monitor, or a TV set hooked up to the color graphics adapter board via an RF (radio frequency) modulator. Any of these options will permit you to operate the examples in this book with one exception: if all you have is the monochrome display, you won't be able to make use of the section on color graphics, in chapter 10. However, if you have any sort of monitor connected to the color graphics adapter board, you will be able to explore both color graphics and character graphics.

The examples used in this book are all based on an 80-column display. With TV sets, and some low-quality color monitors, an 80-column display isn't practical because the screen resolution is so low that the characters get fuzzy; if that's the case then 40 columns must be used. If you're using 40 columns, you will need to do a little mental reformatting to compare the printouts in this book with those on the screen, which will be "wrapped around"; but this should not be a major problem.

Printers

It's very nice but not absolutely necessary to have a printer when writing assembly language programs. Especially as your programs grow longer, looking at a printed listing rather than at the same listing on the screen will be much more convenient and will give you a better idea of the overall operation of your program. Also, when debugging a program, it's nice to be able to look at the listing at the same time you're executing the program and watching the results on the screen.

However, most of the programs in this book are short enough that a printer isn't really necessary. A printer is like a house in the country: if you have one you'll love it, but if you don't you'll get along just fine anyway.

As you become deeply involved in assembly language programming, to the point where you're writing really long programs, then the ideal printer would have more than 80 columns; say 132. This gives you room on your listings for line numbers and extensive comments. Line numbers are a useful addition to long programs because they can be used to create a cross-reference file of symbolic names, as we'll see in chapter 6 when we discuss the CREF program.

One way to get a "wider" printer, if you have an IBM dot-matrix printer or an Epson MX-80 or FX-80, is to set it to "compressed" mode.

(In chapter 4, we show you the techniques you'll need to write a program to do this.) Compressed mode gives you a 136-character width. However, the characters are somewhat harder to read.

Normally the standard 80-column printer is fine. The listings used in this book were originally generated with a standard Epson MX-80 in normal mode.

Documentation

Along with your IBM PC you'll want to have the *IBM Personal Computer Technical Reference* manual, available from IBM. It is packed full of details on the operation of the PC. Many of these details will become important to us as we explore the things that assembly language can do. Also, appendix A of the manual contains a complete listing of the ROM routines built into the computer. After you learn about these routines in chapter 9, you'll find that appendix A will make fascinating reading. (You'll need various other manuals from IBM as well: we'll discuss them in the section on software.)

IBM-Compatible Computers

Many computers claim to be "IBM-compatible," meaning that they will run the same software (and in some cases use the same hardware) as the IBM PC. So do you absolutely have to have an IBM PC to use this book? Maybe not — it depends on which computer you have, since there are various degrees of compatibility.

As a minimum you need a computer that runs the MS-DOS operating system (from which PC-DOS, the system used on the PC, is derived). This way the DOS function calls — which form such an important part of this book — will still apply. However, that's only part of the story. If you want to benefit from the chapter on color graphics, then your computer will have to use the same approach to graphics as the IBM PC does. If you want to understand ROM functions, then the ROM in your computer should operate the same way the IBM PC's does. If you want to use the loudspeaker to generate sound, then your computer will have to do it in a similar way to that of the IBM PC. And so on.

Some computers are compatible in most of these respects, and others in only a few of them. If you have an IBM-compatible computer, you can give this book a try and see how far you get. Most things will probably work. But it takes a detailed understanding of the features of the IBM PC and another particular computer to know in advance how compatible they really are.

Figure I-1 summarizes the hardware needed to use this book.

Software

Let's assume that you have an IBM PC or XT with the necessary peripherals as described previously. What software do you need to use for this book?

The Operating System

For starters you'll need the PC-DOS. At this writing, version 1.10 of this system is included with the IBM PC, and version 2.00 is included with the fixed-disk version of the IBM PC: the XT. For PC owners version 2.00 is available as an option for a very reasonable price.

This book will not be very useful if you are running an operating system other than IBM PC-DOS, such as CP/M-86 or UCSD p-System. Why? Because (among other reasons) our programming examples make extensive use of the specific DOS functions built into PC-DOS. If you use a different operating system, these functions will in most cases be different, and the programs won't work.

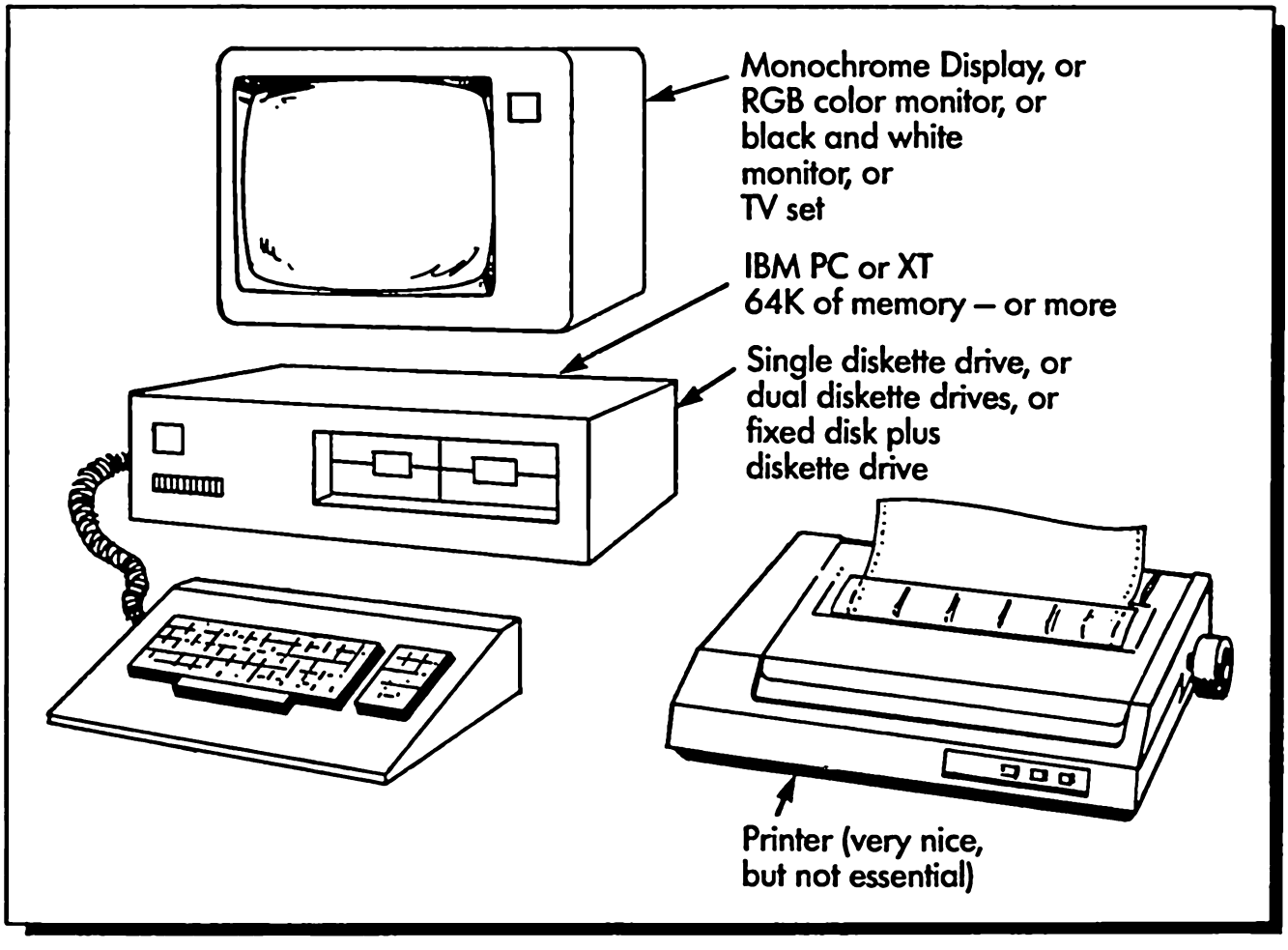


Figure I-1. Hardware needed for this book

Which Version of PC-DOS?

This book will work with any of the current releases of PC-DOS: 1.00, 1.10 and 2.00. We expect that it will also work with any future releases. However, there are some advantages to using version 2.00 (or later), instead of 1.00 or 1.10. First, IBM PC-DOS version 2.00 contains a very useful enhancement to the DEBUG program which is part of the DOS. This is a "mini-assembler," built right into DEBUG. As we mentioned earlier, we will write a number of programs using this DEBUG mini-assembler rather than the more cumbersome ASM (or MASM). It is possible to do this using the older versions of DEBUG that do not have this mini-assembler capability (and we show you how to do it), but it's easier to create the program examples if you have it.

The second reason why IBM PC-DOS version 2.00 is preferable has to do with the way disk files are accessed. Version 2.00 introduces a whole new system of file access, called "file handle access," which we cover in chapter 12. File handle access is a very powerful and flexible system. It uses *pathnames* rather than simple filenames, and is therefore the only system that will work if you have a hard disk drive. Thus if you are interested in learning about this latest file access method, you will need version 2.00.

There are, however, some *disadvantages* to using PC-DOS 2.00. The first is its size. If you have a small amount of memory, like 64K, you will find that version 2.00 takes up so much space that you don't have room for the assembler and assembly-language programs. So if you have a 64K system, stick to DOS version 1.10. This book will work fine with 1.10, except for the slight inconvenience in writing programs in DEBUG, and the inability to perform file handle disk access.

The second disadvantage of 2.00 is compatibility. If you write a program in version 1.10, it will work on version 2.00. However, most version 2.00 programs will not work on version 1.10 or 1.00. If you're writing programs to be used on the the widest possible number of different PCs, then 1.10 should be your choice.

In sum, we recommend that you use PC-DOS version 2.00 if you can. Its increased capabilities, especially the "mini-assembler" in DEBUG, make it well worth the modest price.

DOS Utility Programs

Along with PC-DOS you get a number of utility programs, which are referred to in IBM's documentation as "external routines" (to distinguish them from the functions built right into the PC-DOS program, which are

called "internal routines"). Three of these programs are essential to the use of this book. They are:

1. DEBUG. This program is used to monitor, debug and edit assembly-language programs. Learning to use it, which we teach you in the first few chapters, is vital to an understanding of assembly language.

2. LINK. This program is used to change an intermediate form of assembly-language programs, called OBJ (object) files, into an executable program called an EXE (executable) file. (These terms will all be explained in the following chapters.)

3. EXE2BIN. This program converts EXE files to COM (command) files. COM files are another, somewhat simpler, form of executable program.

Operating System Documentation

Along with the PC-DOS operating system described above, you'll also need the *IBM Personal Computer Disk Operating System* manual which accompanies it. The manual is the definitive word on the operating system, and also on the various utility programs such as DEBUG, LINK, and EXE2BIN. Although we explain how to use these utilities, you will still find the manual important for reference. Also — and this is very important — appendix D of the manual is a list of all the DOS functions available in the operating system. We will explain how to use many of these functions, but for those not covered, and as a reference to all of them, the IBM manual is invaluable.

Assembler Programs

You will need the IBM PC MACRO-Assembler, a software package offered as an option by IBM. This package contains two different assemblers: ASM and MASM. If you have a 64K system you will have to use ASM. If you have more memory you can use MASM, although (as we noted earlier) we recommend using ASM for the examples in this book because of its smaller size and faster loading.

Another program in the MACRO Assembler package which you may find useful is CREF; it produces a cross-reference table of the variable names used in your program.

Of equal importance to the assembler program itself is the *IBM Personal Computer MACRO Assembler* manual which accompanies it. The manual contains a complete list of all the 8088 instructions, a list of all the pseudo-operations used with the assembler, and descriptions of the

various other conventions you'll need to know to use the assembler.

Text Editor or Word Processor Program

In order to create the source files for assembly-language programs you'll need some sort of text edit or word-processing program. If you're a Pascal programmer you're used to this process and you know what "source files" are, but if you've only programmed in BASIC, the idea of preparing a source file may be new to you. Source files (also called ASM files) for assembly-language programs are *text files,* just like letters or other documents. They constitute the first step in the assembly process (unless you're using DEBUG). To create a source file you'll need a word-processing program such as IBM's Personal Editor, WordStar, Easywriter, or any one of the dozens of other excellent programs on the market.

There is a text-editing program which is one of the utility programs that comes with PC-DOS. It's called EDLIN (for EDit LINes). It is possible to use EDLIN to create assembly source files. In fact, it works fairly well for short programs. However, as your programs become longer, EDLIN's limitations become increasingly apparent.

For one thing, EDLIN is a "line-oriented" (as opposed to a "screen-oriented") text editor. This means that you have to specify what line you want to edit, rather than simply move the cursor to that line; this makes it difficult to "move around" in the file. This and other factors make EDLIN suitable only for very short source files.

If you don't already have a good word-processing program, or a full-screen text editor, our advice is to go out and buy one and become familiar with it before you become deeply involved in assembly language. However, it is beyond the scope of this book to recommend a word processor, or to describe how to use it.

The Approach Used in This Book

As we mentioned above, this book is unusual in several respects. The most important of these is that it teaches assembly language in the context of a specific computer: the IBM PC, rather than for all computers using a particular microprocessor. What's so unusual about a book that teaches assembly-language programming for a particular computer? And why is this a superior way to learn programming?

Assembly language consists of instructions *to a particular microprocessor.* The microprocessor chip that powers the PC is the Intel 8088. This microprocessor is the "brains" of the computer. Physically it's very small, consisting of a slice or "chip" of silicon no bigger than your

thumbnail; but mentally it's a giant. The microprocessor interprets instructions you send it in machine language (created by assembly language), and — based on these instructions — causes the computer to do all the things computers do so well: getting data from the outside world, processing it, and outputting it again.

So what would be wrong with a book that taught assembly language for the 8088 microprocessor, without regard to a specific computer? There are a number of books that attempt this approach; they are supposed to work with *any* computer that contains an 8088 or 8086 microprocessor chip. But the fact is, it's very difficult to learn assembly language without reference to a particular computer. There are several reasons for this.

First, while the actual instructions to the 8088 chip may be the same on different computers, *assemblers* (the programs that translate these 8088 instructions into a form the computer can understand) may be different on different machines. So an assembler format that works on one computer may not work on another. If you're reading a book that describes the assembler on machine A, and you're using machine B, then the programs you write may well not run.

Second, there are always a great many differences between computers in such seemingly minor areas as the way the keyboard is used, the format of the screen display, and the operating system commands necessary to accomplish a given task. Since we already know in this book what machine you're using, we can tell you exactly what keystrokes and commands to use to accomplish a given task, such as assembling your program, linking it, and trying it out. No general 8088 book can do that.

There's a third reason it's more effective to teach assembly language for a particular computer rather than for a particular chip. Many computers — including the IBM PC — contain, buried deep within the Disk Operating System (DOS), a collection of routines which can be used by assembly-language programmers to vastly simplify the programs they write. In fact, these routines are so powerful, and such an integral part of today's sophisticated computers, that their use is almost essential for all but the most trivial programs. However, since these DOS routines differ from one machine to another, no book which attempts to teach 8088 assembly language *in general* can make use of them. This book makes extensive use of DOS functions. In fact one of the goals of the book is to teach you everything you need to know to make full use of these powerful software tools.

What it boils down to is this: given the advantages of our approach as outlined above, we think you'll find it easy and enjoyable to learn assembly language from this book.

1

Assembly Language and Debug

Concepts

Assembly language versus higher-level languages
Using DEBUG
Memory
Memory addressing
ASCII codes

Debug Commands

D = Dump
F = Fill

In this chapter we're first going to talk about assembly language in general. We'll explain how it differs from higher-level languages such as BASIC or Pascal, and talk in a general way about the operation of an *assembler* and how it differs from the interpreter or compiler used in higher-level languages.

In the second part of this chapter we'll introduce you to DEBUG, the utility program which will be your gateway into assembly-language programming.

Assembly Language and Higher-Level Languages

As is true with most computer languages, it's hard to describe assembly language meaningfully without reference to examples of specific programs. In the next chapter you'll encounter your first assembly-language program, and then you'll begin to see what assembly language

is all about. In the meantime, we'll provide an overview concerning what assembly language is, and how it differs from other computer languages.

Higher-Level Languages — More Abstract

If you are familiar with a higher-level language such as BASIC or Pascal, you know that there is a certain level of abstraction involved in program statements in these languages. A BASIC statement such as

```
LET A = 3
```

or, in Pascal,

```
A := 3
```

is operating on an abstract level in that we don't usually know, or *need* to know, where in the computer the "A" is, or what changes are taking place in the computer when A is assigned the value 3. This is because higher-level languages are oriented toward the handling of numbers with algebra-like formulas. Thus FORTRAN, one of the earliest of the higher-level languages, stands for FORmula TRANslator — a language in which it is easy to express formulas. BASIC is a descendant of FORTRAN, and it too — as is Pascal — is oriented primarily toward processing numerical data in this abstract, algebraic context. Programmers in these languages *want* to be insulated from what's really going on inside the computer so they can concentrate on the formulas.

Analogy — a Newspaper Office

As an analogy to a higher-level language, we can think of a newspaper office. Reporters write stories about the affairs of the day: an election in Pennsylvania, a flood on the Mekong River, a riot in Bombay. This information is all transmitted to the newspaper office. There it is edited, typeset, pasted up, printed, and finally distributed to newsstands and tossed by small children into people's driveways.

The data processed by the newspaper is abstract. Although you can touch the medium (paper) that contains it, you can't touch the actual news: you can't build houses out of headlines or drive gossip to work. The value of news lies in the information itself.

In a similar way a computer program written in a higher-level language is concerned with something abstract: variables representing numbers and characters.

Assembly Language — More Concrete

In contrast, assembly language operates on a very concrete level. It deals with bits, with bytes, with *words* (two bytes side-by-side), with *registers* — which, as we'll see, are physical places in the microprocessor where bytes and words are stored — and with memory locations, which have specific numerical addresses and specific physical locations in the memory chips inside the PC.

An analogy to assembly language might be a brick factory. In this factory, clay, water, and energy to run the kilns are the raw materials. The factory performs certain operations on these raw materials, and the output from the factory is the bricks themselves, packaged in bundles which can be lifted onto trucks by forklifts and delivered to building sites.

You can touch a brick, but you can't touch a news story. Similarly, you can (or could, if you were very small) touch the registers and memory locations that assembly language deals with, while you can't touch the variable "A" in a BASIC program. (See Figure 1-1.)

The General and the Specific

If you travel to another city, you will have to buy another newspaper, but the news will be much the same. We could say that the "program" — the series of operations used to generate the news — is similar in most newspapers. Higher-level languages are similar in that they can run on a variety of different computers: the BASIC program on my IBM PC will

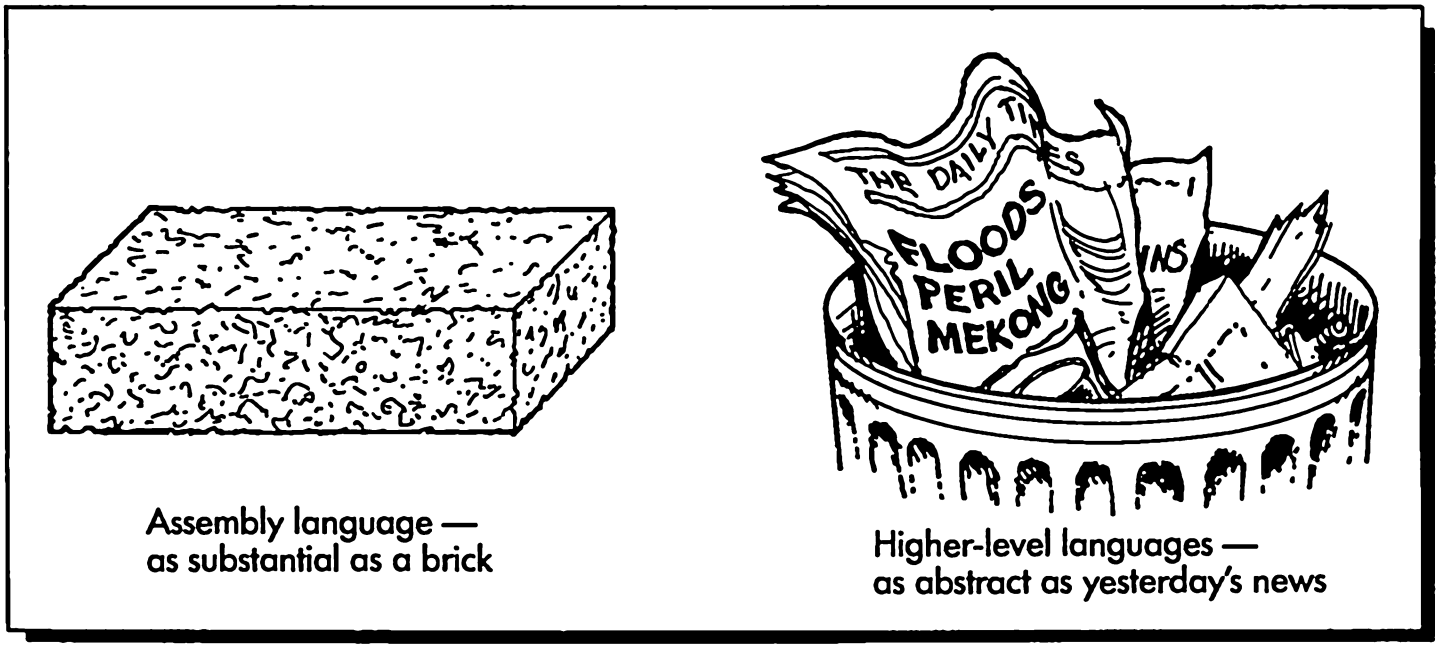


Figure 1-1. Assembly language and higher-level languages

probably — with some minor modifications — run on someone else's Apple.

In the brick factory, on the other hand, the operations are much more specific. The clay must be dumped into tanks, the water must be mixed in, and the kiln must be heated to a certain temperature. These procedures are applicable only in one particular factory. If the foreman says to turn up the temperature of kiln number five to 2000 degrees, this instruction is tailored specifically to the physical equipment of one particular factory. In a similar way, programs written in assembly language are specific to a particular microprocessor chip, and in many cases to the specific computer which contains the chip as well.

What Does an Assembler Do?

If you've written programs in BASIC, you're familiar with the two-step process involved: first you write a group of BASIC program statements which make up a program; then later, when you execute the program, these statements are "interpreted," or changed into *machine-language* instructions which are executed by the 8088 microprocessor. (We'll have a lot more to say about machine language in the following chapters. Don't worry if you don't completely understand what we mean by it at this point.)

You may not be very aware of this interpretation process in BASIC, since it is made to appear "invisible" to the user; but it takes place nevertheless. The individual program lines are interpreted one at a time, and the resulting machine-language instructions for each line are executed by the 8088, before the next line is interpreted. (Refer to Figure 1-2 for a simplified view of this process.)

In compiled languages such as Pascal, things are handled a little differently. The user first creates a *source file*, which is a text file of the entire program. This is then changed into machine-language instructions by a *compiler* program. (Actually a linker is used too, but we'll ignore it for the moment.) In a compiled language such as Pascal, the entire program is transformed into machine language at once.

Assembly language resembles a compiled language more than it does an interpreted language such as BASIC. An assembler source file consisting of the text of the program is first created. This is then *assembled* into machine-language instructions by an *assembler* program. The *assembler* performs a process very similar to a *compiler*, except that — as we'll see in the next chapter — there is a far closer correspondence between an assembly-language instruction and a machine-language

instruction than there is between a Pascal statement and the resulting group of machine-language instructions.

What we've described is the traditional way of transforming an assembly-language program into machine-language instructions. However, in the first few chapters of this book we'll use a different approach: a feature of the DEBUG program called a "mini-assembler." Using DEBUG it's almost as easy to create and run short assembly-language programs as it is to create and run interpreted programs such as with BASIC. We'll introduce you to DEBUG later in this chapter, and show you how it can be used to assemble a program in chapter 2.

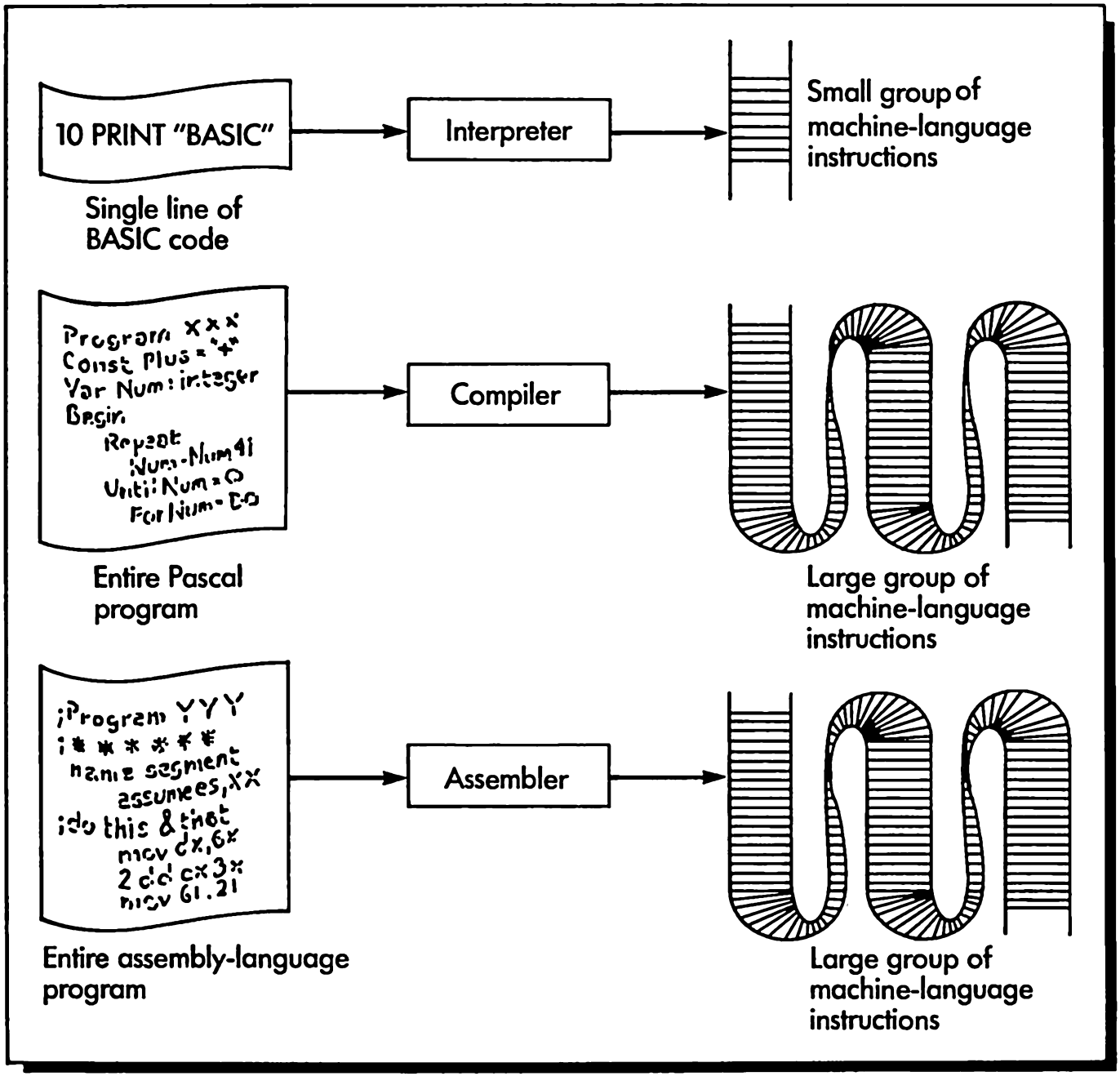


Figure 1-2. Interpreters, compilers, and assemblers

Microprocessors

We've mentioned microprocessors several times. A microprocessor is a single chip of silicon which performs all the basic functions of a computer. Because assembly language is inextricably entwined with a particular microprocessor chip — the Intel 8088 in the case of the IBM PC — we'll talk a bit here about the 8088 and its history. Figure 1-3 gives a representation of the 8088's development.

The very first microprocessor was the 4004, manufactured by the Intel Corporation. It appeared in 1970. Before the 4004, computers were made differently. The earliest solid-state computers had thousands of individual transistors mounted on hundreds of printed circuit boards,

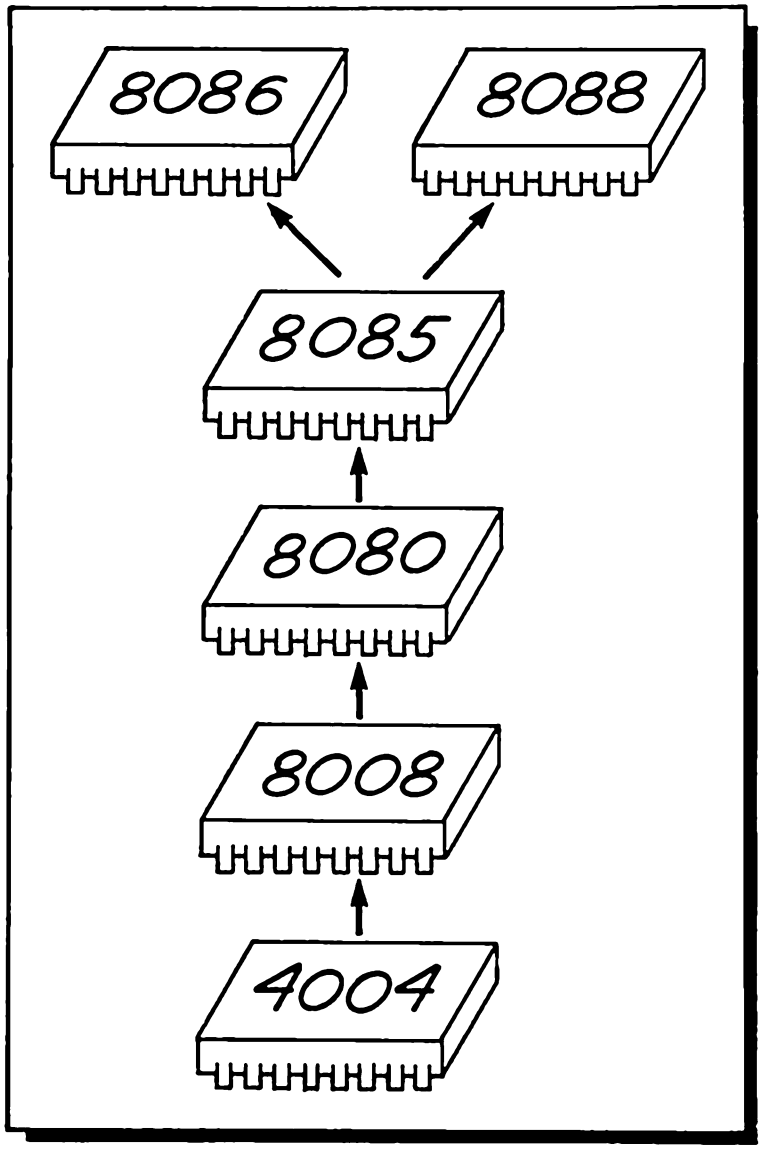


Figure 1-3. 8088 family tree

which occupied enormous cabinets in air-conditioned rooms and cost hundreds of thousands of dollars. Later, integrated circuits — which put a dozen or more transistors in a little package — reduced the size of a computer to somewhat smaller cabinets in rooms that weren't necessarily air-conditioned, but the computer still cost in the six or even seven-figure range.

The 4004, in what is surely one of the most astonishing accomplishments of our age, squeezed all these cabinets into an object so small it would blow away if you sneezed, and cost (in quantity) less than a good dinner.

The 4004 was not really a very powerful microprocessor. It operated on data which was only 4 bits wide, and had a rather rudimentary instruction set. But it was followed soon after by the first 8-bit microprocessor, the 8008. The 8008 evolved into the 8080 (a much easier to use 8008), and then into the 8085, a refined 8080, which is still in use in millions of computers.

The next major advance was to go from eight bits to sixteen bits, since 16-bit microprocessors provide more power and the capability to use a larger memory than do their 8-bit cousins. The microprocessor that achieved this breakthrough was the Intel 8086. The 8086 operates on 16-bit data: it requires a 16-bit memory, 16-bit data buses (which connect the components of the computer system together), and other 16-bit peripheral devices.

However, because 8-bit computers have been around for so long, many of these peripheral devices exist at a reasonable price only in 8-bit form. So Intel created another version of the 8086, which it called the 8088. The 8088 has an internal architecture like the 8086: the same 16-bit registers. But when it talks to the outside world, it does so with 8-bit data: one byte at a time. Thus the memory and peripherals used with an 8088 can be the tried and true (and cheaper) 8-bit models. This reduces the cost of the computer system, and is the approach used in the IBM PC.

DEBUG Versus the Assembler

As we noted above, there are two major ways to write short assembly-language programs on the IBM PC. The first way is to use the assembler program ASM, or its more sophisticated cousin MASM. (We explained the difference between these two programs in the Introduction.) People

usually write assembly-language programs in one or the other of these assembler programs. (Yes, we know you may not be entirely clear at this point what an assembler program is supposed to do. That's all right — we'll get to it soon.)

The other way to write assembly-language programs is to use a different kind of program called DEBUG. DEBUG is not really an assembler program. Its primary use is for "debugging" (that is, fixing the errors in) assembly-language programs. However, you can also write short assembly-language programs with DEBUG.

We've chosen to write the programs in the first few chapters of this book using DEBUG. There are several reasons for this. First, DEBUG is a much easier program to operate than the ASM (or MASM) assembler program. To type in and execute a program using DEBUG requires calling up only DEBUG itself: a simple process. Using an assembler, on the other hand, involves using a text editor, the assembler itself, a program called LINK, and often another program called EXE2BIN. Each of these programs requires a rather complex series of commands to make it work. We figured you'd have enough on your mind being introduced to a new computer language, without having to learn how to operate all these other programs at the same time.

DEBUG's second advantage is that programs written with it require less "overhead" than those written with the assembler. This overhead comes in the form of program statements which must appear in the ASM "source file," but which are not necessary in DEBUG. (Don't worry if you don't understand what we mean by "source file"; we'll explain everything eventually.) By using DEBUG you avoid having to start your day with a lot of incomprehensible program lines that would be necessary in the assembler.

Third, using DEBUG puts you in closer contact with what is *really* going on in your computer than using the assembler would. As we'll soon see, DEBUG has features that make it possible to get down to the most fundamental level of your computer's operation (short of opening up the cover and probing about with meters and oscilloscopes). Sooner or later, if you write programs in assembly language, you're going to have to understand this fundamental level and learn to use DEBUG; so now seems like a good time to start.

Of course, as we'll find later, the assembler has all sorts of powerful features that make it indispensable for assembling long programs, but for the moment, DEBUG will do just fine. The table below summarizes the advantages and disadvantages of DEBUG and the assembler.

DEBUG versus Assembler

DEBUG	Assembler
Easy to run	Hard to run
Low overhead programs	More program overhead
Close to the machine	Isolated from the machine
Not so versatile	Very versatile
Good on short programs	Good on long programs

The Window of the 8088's Soul

An old saying has it that "the eyes are the windows of the soul." We might say that DEBUG is the window of the 8088's soul. Besides being useful for assembling programs, DEBUG is also used to examine and modify memory locations; to load, store and start programs; and to examine and modify registers (we'll learn what "registers" are later). In other words, DEBUG is designed to put us in touch with various physical features of the IBM PC.

Before we write our first 8088 assembly-language program in the next chapter, we're going to get to know our way around DEBUG: rev it up, so to speak, find out where the controls are, and taxi it out of the hangar and around the runway. Then we'll be ready for takeoff in chapter 2.

Getting DEBUG Rolling

All right, let's leap into the cockpit, get a firm grip on the keyboard, and get DEBUG rolling! We'll assume that you have a disk with DEBUG on it inserted in drive A, and that the A> prompt is waiting for your next move. (If you have a fixed disk you'll have to make sure DEBUG has been copied to the fixed disk, and you'll also have to imagine a "C>" whenever you see an "A>" in the text of this book.) As we noted in the introduction, DEBUG is one of the programs provided on the "system disk" that contains the PC-DOS.

Following the DOS prompt, enter the program name "DEBUG".

(When we tell you to "enter" something in this book we mean to type the "something" and then press the ↵ key — the one just to the left of the numeric keypad.)

```
A>debug          ←— Enter this
-                ←— DEBUG's prompt character
```

The single dash that appears on the screen is DEBUG's "prompt," the symbol it uses to tell you that it's ready to listen to what you have to tell it.

The "D" Command

You tell DEBUG what to do by typing in single-letter commands, usually followed by one or more numbers. When we refer to these single-letter commands in the text we usually use uppercase letters to make them stand out (like "D"). However, when you type them in, you can use lowercase. It works just as well as uppercase, and is easier to type. For example, enter the letter "d", followed by the digits "1", "0", and "0".

```
-d100            ←— You enter this

08F1:0100  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0110  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0120  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0130  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0140  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0150  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0160  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
08F1:0170  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

Wow — look at all those numbers! What does it all mean? Well, first of all, you may not see all zeros on your display as we show here. What the "D" command has done is to "dump," or display, a portion of your computer's memory on the screen. Each pair of numbers represents one byte, or eight bits, of data stored in a particular memory location. If your computer's memory happened to have other data in it before you loaded DEBUG, it will appear here when you type "D", so you may see all sorts of junky-looking numbers, like this:

```
-d100

08F1:0100  03 EB 42 90 75 03 EB 41-90 2C 30 72 38 3C 0A 73   .kB.u.kA.,0r8<.s
08F1:0110  34 52 8B D3 9F 03 DB 03-DB 03 DA 03 DB D1 DE 9E   4R.S..[.[.Z.[Q^.
08F1:0120  D1 D6 8A D0 B6 00 9F 03-DA D1 DE 9E D1 D6 5A E8   QV.P6...ZQ^.QVZh
```

```
Ø8F1:Ø13Ø  21 ØØ 72 ØA 74 ØA 2C 3Ø-72 Ø4 3C ØA 72 D3 41 4A   !.r.t.,Ør.<.rSAJ
Ø8F1:Ø14Ø  8A C7 ØA CØ C3 9F 41 4A-9E F9 C3 E8 Ø5 ØØ 75 Ø1   .G.@C.AJ.yCh..u.
Ø8F1:Ø15Ø  C3 EB F8 8A C5 ØA C1 75-Ø1 C3 49 42 8B F2 AC 24   Ckx.E.Au.CIB.r,$
Ø8F1:Ø16Ø  7F ØA CØ F9 75 Ø1 C3 3C-ØC F9 75 Ø1 C3 3C ØA F9   ..@yu.C<.yu.C<.y
Ø8F1:Ø17Ø  75 Ø1 C3 3C 1A F9 75 Ø1-C3 72 Ø1 C3 F5 C3 A9 46   u.C<.yu.Cr.CuC)F
```

All the numbers in this display are in hexadecimal. In fact, hexadecimal is the only numbering system that DEBUG knows about, so if you aren't already acquainted with this way of representing numbers, now is the time to read appendix A in the back of this book.

(Welcome back, those of you who have been reading appendix A.) Let's adopt this convention: hexadecimal numbers — except those in program listings or where the context makes clear what they are — will be followed by a small letter "h" to distinguish them from decimal numbers. Decimal numbers — again, unless the context makes it clear — will be followed by a small "d". Numbers from 0 to 9 are the same in both systems, so they don't really need to be followed by a distinguishing letter, although they sometimes are for consistency. Of course, since DEBUG *only* speaks hexadecimal, it doesn't use an "h" in its printouts, and you don't need to put one after hexadecimal numbers you type in as DEBUG commands.

As you know, it requires two hexadecimal digits to represent an 8-bit byte of data. This two-digit hexadecimal number can range in value from 00h to FFh (which is from 0 to 255d). Thus all the two-digit numbers in the printout above fall into this range. There are 16d of these numbers on each line of the display. The dashes in the middle of the printout are placed there for clarity, to separate the left-hand eight bytes on the line from the right-hand eight bytes.

Addresses

The numbers in the column to the left (like 08F1:0120) are the *memory addresses* of the bytes of data. Thus each byte shown in the dump occupies a specific address, as shown in Figure 1-4.

The vertical column to the left in Figure 1-4 represents an actual section of your computer's memory. Notice how each memory location, or *byte*, corresponds to a particular number in the DEBUG dump.

Each address consists of two numbers separated by a colon. What do these two numbers mean?

Offset Address

The 0100 part of the number, to the right of the colon, is called the *offset* address. For the next few chapters this will be the only part of the

address we'll be concerned with, so if you want to skip the next few paragraphs it won't really do you any harm.

Segment Addresses

The 08F1 part of the number, to the left of the colon, is the *segment* address. (Your system might have a number other than 08F1. That's fine too.) The segment part of the address is such a complex and far-out thing that we're going to postpone a thorough discussion of it until chapter 8. However, we'll tell you here in very general terms what it means, so you won't have to wonder about it for six more chapters.

> To find a real address, take the segment address, shift it left one place, and add the offset address.

Briefly, the idea of the two-part address is this. The 8088 operates mostly on numbers which are *four* hexadecimal digits wide, like FFFFh or 1234h. (We'll abbreviate the word "hexadecimal" to "hex" from now on.)

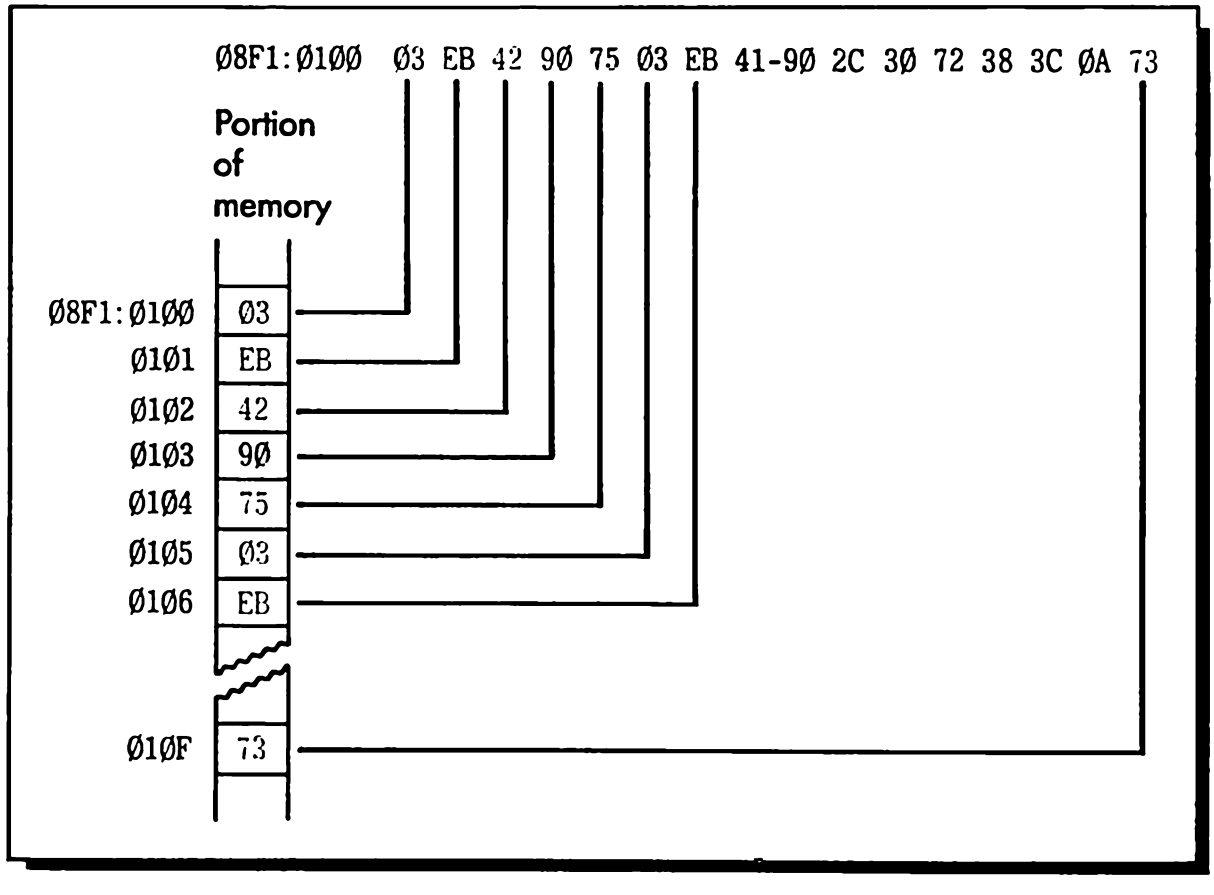
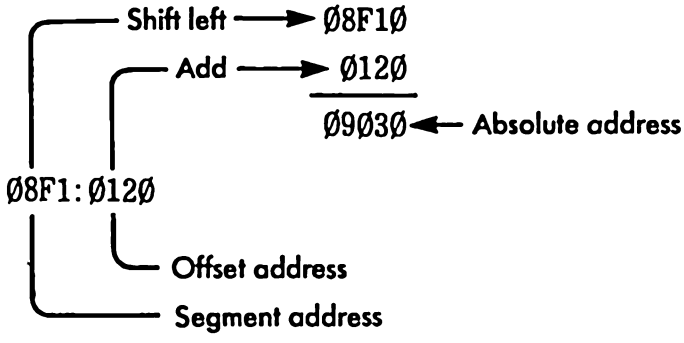


Figure 1-4. Each byte in the dump is a byte in memory

However, there are so many possible memory addresses in the 8088 that it takes numbers with *five* hex digits to specify them, such as FFFFFh or 12345h. The engineers at Intel invented the following solution to this dilemma. They used *two* four-digit hex numbers to represent each memory address: the first number is the offset address and the second is the segment address. These numbers are combined in an unusual way to form the real or *absolute* address. The *segment address* (the number on the left) is shifted left one digit — which is the same as multiplying it by 10h. It is then added to the *offset address* (the number on the right).

For example, suppose an address shown in our DEBUG dump is 08F1:0120. What absolute address do these numbers represent? We take the 08F1 and shift it left to get 08F10. Then we add the 120. The resulting five-digit sum is the hex number representing the absolute address of this particular memory location, as shown below:



From now until chapter 8 we're not going to be concerned with the segment part of the address. How is this possible? The reason we can get away with paying attention only to the offset part of the address is that we're going to operate only in a certain part of memory: a part called a *segment*. This part of memory is 64K bytes long, which is 65536d bytes, or FFFFh bytes. It can be specified with a single four-digit hex number, so all we need to specify an address in the segment is the four-digit offset address. If this isn't completely clear, trust us. It will all be explained in chapter 8, on memory segmentation.

Offset Addresses and DEBUG

Notice how each offset address (we'll just call them "addresses" now, at least until chapter 8) in the left-hand column of a DEBUG "dump" ends with a zero. If you're familiar with hex numbers you should understand why this is so. There are 16d, or 10h, bytes in each line, so when you've counted from 0h to Fh, you're ready to increase the ten's column by 1, since 10h is the number that comes after Fh in hex. So we

display 16d (10h) bytes, and then move down one line, increment the address by 10h, and display 10h more bytes.

The display would be easier to read and understand if it had the one's column values of the addresses printed across the top, like this:

```
          Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
Ø8F1:Ø1ØØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø11Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø12Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø13Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø14Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø15Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø16Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø17Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
```

But it doesn't. Anyway it should be clear that the first byte on the top row is at memory location 0100, the next is at 0101, the next at 0102, and so on. Similarly, the first byte on the second row is at 0110, the second at 0111, and so on.

The "F" Command

Want to see this display change? An easy way to do that is to use DEBUG's "F" or "fill" command. This command fills a part of memory with a particular hex number. To use "fill" you enter "f" followed by three numbers, each number separated by a space. The first of these numbers is the address where you want to *start* filling, the second is the address where you want to *stop* filling, and the third is the constant (from 00h to FFh) that you want to use to fill in between the first address and the second. Notice that while the data to be filled in consist of two-digit hex numbers (bytes), the addresses are four-digit hex numbers. Of course, you don't need to type leading zeros, so you can type fewer than four digits for the addresses when appropriate, as it is here.

Enter this:

```
-f12Ø 14f ff
```

Constant to be filled in

Ending address

Starting address

Nothing will appear to happen. To see what's changed, you have to dump the same part of memory again:

```
-d1ØØ
Ø8F1:Ø1ØØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø11Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø12Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
Ø8F1:Ø13Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
Ø8F1:Ø14Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
Ø8F1:Ø15Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø16Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø8F1:Ø17Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
```

Well, look at that! All the memory locations between 120 and 14F are now filled with FF, just as you specified with the "F" command. (Of course if you started off with other numbers instead of zeros, they'll still be there instead of the zeros shown in this dump.)

ASCII Codes

You may have been wondering about all the little dots and odd characters on the right-hand side of the dump display. These are the characters (like "A", "B", and so on) that the numbers to the left represent. The number which represents a particular character is called its "ASCII Code." (ASCII stands for "American Standard Code for Information Interchange.") As you probably know, the ASCII code is the normal way to represent characters in a computer's memory. (There is a very nice table of these codes in the *IBM Personal Computer Technical Reference* manual.)

Since neither 00 nor FF represents a printable ASCII character, the positions in the ASCII display corresponding to these numbers are filled with dots, which indicate "no printable character." (If your computer's memory was filled with junk rather than all zeros to begin with, some of the numbers may have been printable characters.) To see the character display change, along with the numbers, let's try filling in parts of memory with numbers that we know represent printable ASCII characters.

Enter the following DEBUG commands:

```
-f1ØØ 117 61
-f178 17f 24

-d1ØØ
Ø8F1:Ø1ØØ  61 61 61 61 61 61 61 61-61 61 61 61 61 61 61 61  aaaaaaaaaaaaaaaa
Ø8F1:Ø11Ø  61 61 61 61 61 61 61 61-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  aaaaaaaa........
Ø8F1:Ø12Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
Ø8F1:Ø13Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
Ø8F1:Ø14Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF  ................
```

```
Ø8F1:Ø15Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø8F1:Ø16Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø8F1:Ø17Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-24 24 24 24 24 24 24 24   ........$$$$$$$$
```

61 hex is the ASCII code for a lowercase "a," and 24 hex is the code for the dollar sign ($). We can see them both as numbers, and, to the right, as characters.

Summary

In this chapter we've talked about how assembly language differs from higher-level languages, and also explained something about the operation of the DEBUG utility program. You may find it useful at this point to experiment a bit with DEBUG. Try filling in different constants to see how they look when you "dump" them. Examine different parts of memory. You'll be using DEBUG a great deal in the chapters to follow, and you should feel comfortable about using it.

2

Instant Program

Concepts
- Writing a simple program in machine or assembly language
- Assembly language instructions

Debug Commands
- E = Enter
- A = Assemble
- U = Unassemble
- G = Go

8088 Instructions
- MOV = Move
- INT = Interrupt
- JMP = Jump

DOS Functions
- Display Output
- Program Terminate

In this chapter we're going to start by describing the writing of a complete, though very short, assembly-language program. Then we'll go back and talk in more detail about the steps used in the process and what they mean. We'll finish up with some variations on our program.

Writing Your First Program

You're going to write your first program in assembly language, but you don't know assembly language yet. Obviously, there will be many aspects of the process that won't seem completely clear to you. Don't worry! Our approach is to show you first *what* something looks like, and afterwards explain *why* it looks that way and *how* it works.

By moving in this direction, from the concrete to the abstract (rather than the other way around), we hope to avoid the sort of academic theory-oriented descriptions that leave most readers confused, bored, and frustrated. Instead, you'll first get the *feeling* of the process (the roar of the motor and the rush of the wind in your hair, to return to our flying analogy). Later we'll explain what happened.

The Two Versions of DOS

There's a small problem we better deal with right away. This has to do with which version of DOS you're using. As we noted in the introduction, the DEBUG in DOS version 2 (that is, versions 2.00 and later) contains a built-in mini-assembler which will help in the creation of assembly-language programs. The DEBUG in DOS version 1 (versions 1.00 and 1.10) does not have this capability, so for those readers using this version we need to take a slightly different tack.

We'll handle this situation in the following way. We'll first explain how to type in a program if you're using DOS version 1. Even if you have version 2, you should read this part, try it out, and understand it. There are two reasons why this is a good idea. The first is that you will be introduced to a new DEBUG command: the "E" (for "Enter") command. The second is that after you've typed in the program using "E", you'll be better able to appreciate how lucky you are to have DOS version 2, with its advanced version of DEBUG and its mini-assembler capability.

Writing the Program with the "E" Command

In this section we'll create an assembly language program using DEBUG's "E" command. (The term "assembly language" is actually not quite right in this particular instance, as we'll see later in the chapter, but that needn't concern us now.) If you have DOS version 1 this is the *only* way to use DEBUG to create a program. If you have version 2, you should, as we suggested above, follow along anyway, typing in the commands we show.

The purpose of the "E" command is to *enter* a byte (or bytes) of data into memory. It's a little like the "F" command described in the last chapter, except that you can enter a series of *different* bytes; they don't all have to have the same value, as "F" required.

The series of bytes we're going to enter with "E" will constitute our program. To insert this program into memory, you enter the "E" command, followed by the address where you want the program to go. In our case, we're going to put it at location 100h, so we enter "e" followed

by "100". The program will respond by printing out the address, followed by its current contents:

```
-e1ØØ              ← You enter this
-Ø4B5:Ø1ØØ 61._ ← The cursor will wait here for you to type in a two-digit number
   ↑      ↑ Offset address is the same as what you typed in
   Segment address doesn't matter
```

As shown here the contents of location 100 happen to already be 61h, since that's what we put there with the "F" command earlier. However, it doesn't really matter what was there before: the important thing is what we're going to put there now.

To "enter" a two-digit hex number into this location we type the number followed by the *space bar* (*not* the ↵ key). The space bar has the effect of entering *one number* and going on to the next one. The ↵ key, on the other hand, enters the number and then terminates the entire "E" command sequence and returns us to the DEBUG prompt. After you have entered a number and pressed the space bar, the command will then print the old contents of the *next* location, and wait for you to type in the new contents.

The series of hex numbers we want to type in is the following:

```
B2
1
B4
2
CD
21
CD
2Ø
```

These are the numbers that constitute our program. Type each number, press the space bar, type the next number, and so on. After you've typed in all 8 numbers, the screen should look like this:

```
-e1ØØ
Ø4B5:Ø1ØØ  61.b2   61.1    61.b4   61.2    61.cd   61.21   61.cd   61.2Ø
-
↑                                                                  ↑
DEBUG prompt       Press space bar following these numbers         Press ↵ following last number
```

After we type the last number, we press ↵ to tell DEBUG that we're through. This should cause the reappearance of the DEBUG prompt (–). If you don't type any number at all before hitting the space bar, the byte in that location will remain unchanged, as you may discover if you make a typing mistake. If you make a mistake, just press ↵ to get back to the DEBUG prompt, and start over.

You've now placed your program in the computer's memory, from location 100 to 107, using the "E" command. We'll explain how to execute or "run" the program in a moment.

Writing the Program with the "A" Command

Here's where you DOS version 1 users become, briefly, mere spectators. You should read through this section so you know what you're missing, and more importantly, because program descriptions in the next few chapters are going to be based on the "A" command approach outlined here. You'll need to know both approaches, so you can use "E" even though we're talking about "A"; that is, translate our descriptions of the "A" approach into operations with "E". This won't be as hard as it probably sounds; so read on. Or, better yet, hurry out and buy a copy of DOS version 2.

The "A" command accomplishes the same thing as the "E" command — that is, putting the bytes which constitute our program into memory — but it does it in a different way. When we use the "A" command we don't insert hex bytes into memory directly. Instead, we type in a series of "mnemonic symbols." ("Mnemonic" simply means "easy to remember.") These symbols are supposed to be easier to remember than the hex numbers they represent. They are two- or three-character names which stand for certain assembly-language *instructions*. An instruction tells the microprocessor *what operation is to be done.* It is usually followed by a space and then by some letters and numbers that indicate what the operation is to be done *to.*

Expressed in mnemonic symbols, our program looks like this:

```
mov dl,1
mov ah,2
int 21
int 2Ø
```

Looks short, but absolutely incomprehensible, doesn't it? That's all right, it won't be long before you can churn out this kind of thing in your sleep. We're going to type in this program, then dissect it a little and see if we can get a feel for how the "A" approach relates to the "E" approach,

and for what assembly language is all about.

Enter the letter "A" followed by the address where you want the program to go. Here's a rule you should remember: *Programs written in DEBUG should always start at 100h.* The reasons for this will become clear later, when we talk about the difference between COM files and EXE files.

> Programs written in DEBUG should always start at offset address 0100h.

When you enter "A" followed by an address, DEBUG will automatically echo the address:

```
-a1ØØ
ØBF1:1ØØ-
```

The cursor will wait here, blinking

DEBUG will then sit there waiting for you to type in the mnemonic codes for your program. Enter "mov dl,1" on the first line. That's "mov" as in the first three letters of "move," a space, which is important, then "dl,1" follow. Don't confuse letters and numbers: it's the *letters* "dl", followed by a comma and the *number* "1". The screen should now look like this:

```
ØBF1:Ø1ØØ mov dl,1
ØBF1:Ø1Ø2 _
```

You've typed your first line of assembly language! You should reward yourself, in a modest sort of way (with a cookie, or perhaps a swig of Scotch).

The assembler is waiting for line two. Enter "mov ah,2". Then the third line, "int 21", and the fourth, "int 20". After you've finished these four lines, you're done. So when the program tells you

```
ØBF1:Ø1Ø8
```

you simply press ↵ to let it know you're through assemblying this program and want to get back to DEBUG's prompt.

Your screen should now look like this:

```
-a100
08F1:0100 mov dl,1
08F1:0102 mov ah,2
08F1:0104 int 21
08F1:0106 int 20
08F1:0108
-_
```

← Enter these lines

← Simply press ENTER here

← Now you're back in DEBUG

So, you now have two different ways to enter your program, depending on which version of DOS you have. In either case the program itself should now be sitting in memory, waiting to be executed. There's a lot to say about the relationship between these two approaches to putting the program into memory, but if you're a real red-blooded programmer you can't wait to run the program. So let's do that first, and talk later.

Running the Program

What does this program do? Does it balance your checkbook? Calculate accounts receivable? We're afraid it's not quite so ambitious as that. Let's see what happens when we run it. To execute the program we use the "G" (for "Go") command. Simply enter the letter "g". It's not followed by any numbers (this time). This will cause the program to be executed, just as entering RUN does in BASIC.

```
-g
☺
Program terminated normally
-
```

← Enter this

What happened? The program printed a happy face on the screen! If you didn't get a happy face, you probably made a mistake typing in the program. It's easy to mistype something, with all the numbers and unfamiliar symbols. Start over with the "E" or the "A" command, and try again.

Unfortunately mistakes in assembly-language programs can have more serious consequences than those in higher-level languages like BASIC. In higher-level languages the interpreter or compiler usually

protects the operating system from the consequences of errors in programming, so that your machine keeps running and says something like "Error in line 2034."

In assembly language, however, there is no such protection. Assembly language is the most fundamental level of the machine: there is nothing on a "supervisory level" overseeing the assembly-language program, as the interpreter or compiler does in higher-level languages. So if you make a mistake in assembly language it is woefully easy to "crash" your operating system — that is, alter parts of it in memory so that it no longer works and you need to reset the entire computer, either by hitting the Alt, Ctrl and Del keys simultaneously, or, in even worse cases, by turning the entire computer off and then on again. But all this is academic — you're never going to make a programming error, are you?

Let's look in memory with the "D" command and see if we can find our little program.

```
                    Program ──────────────────────────────────┐
-d1ØØ                                                         │
Ø8F1:Ø1ØØ  B2 Ø1 B4 Ø2 CD 21 CD 2Ø-61 61 61 61 61 61 61 61   2.4.M!M aaaaaaaa
Ø8F1:Ø11Ø  61 61 61 61 61 61 61 61-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   aaaaaaaa........
Ø8F1:Ø12Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
Ø8F1:Ø13Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
Ø8F1:Ø14Ø  FF FF FF FF FF FF FF FF-FF FF FF FF FF FF FF FF   ................
Ø8F1:Ø15Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø8F1:Ø16Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø8F1:Ø17Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-24 24 24 24 24 24 24 24   ........$$$$$$$$
```

There it is, in the first 8 locations, from 100 to 107, as you can see by comparing these numbers with those typed in using "E". Our program has overlaid the 61s that were there before.

The symbols on the right, 2.4.M!M, are meaningless. They just happen to be the ASCII equivalents of the numbers that make up the program.

What an Assembler Really Does

If you typed in the program using "E" you probably aren't too surprised to see these numbers reappear when you look in memory with "D". After all, you entered the numbers into memory, and there they are, just where you put them.

But how did they get there if you used the "A" command? You typed in mnemonic instructions, and lo and behold, there are hex numbers

sitting in memory! What's happened here is what this book is all about: the "A" command *assembled* the mnemonic instructions into hex numbers. This is the function of an assembler; both of DEBUG's mini-assembler invoked with the "A" command, and of its large-scale relatives, the ASM and MASM assemblers. We'll have more to say about this in a moment. First let's look at our little program from another perspective.

The "U" Command

There's a more elegant and useful way to look at our program than by using "D" as we did above: the "U" command. We've assembled our program with "A", or typed in the pre-assembled hex numbers with "E". Now let's use the "U" command to *unassemble* it. "U" is the opposite of the "A" command. Where "A" takes us from symbolic mnemonic codes to the hex digits of machine language, "U" takes us from hex digits back to mnemonic codes. (Actually the usual word for "unassemble" is "disassemble." But, since the "D" command was already taken, IBM decided to use "unassemble." We'll use both words in this book, as the spirit moves us.)

To "unassemble" your program, enter "U", followed by the address where you want to start disassembling, then a comma, and then the address where you want to *stop* disassembling, like this:

```
-u100,106      <-- You enter this

08F1:0100 B201        MOV     DL,01
08F1:0102 B402        MOV     AH,02
08F1:0104 CD21        INT     21
08F1:0106 CD20        INT     20
-_
```

<-- The program prints out all this!

There's the program in *both* hex codes and mnemonic instructions, all nicely arranged for you to admire! Thus the hex number B201 is the machine-language equivalent of the assembly-language statement "MOV DL,01" and so on for the other instructions. As before, the numbers on the left, such as 08F1:0100, are the addresses of the locations occupied by the program. There's an address printed for each instruction, and since each of the instructions happens to occupy just two bytes, the addresses are all even numbered: 100, 102, 104 and 106.

Machine Language and Assembly Language

The hex numbers on the left in the "U" listing above are what is

called *machine language.* These numbers occupy specific memory locations, and the 8088 microprocessor looks in these locations, takes the numbers out of them, figures out what they mean, and executes them. These numbers are called "machine language" because it's the *machine* — the microprocessor — that understands them and operates on them. As far as we humans go, such numbers are hard to understand and hard to remember. For a human to decipher a program written entirely in hex numbers requires the most masochistic form of mental discipline, while the microprocessor chip, no larger than a pea, handles it easily. It is perhaps better not to dwell on the philosophical implications of this.

```
Ø8F1:Ø1ØØ B2Ø1          MOV     DL,Ø1
Ø8F1:Ø1Ø2 B4Ø2          MOV     AH,Ø2
Ø8F1:Ø1Ø4 CD21          INT     21
Ø8F1:Ø1Ø6 CD2Ø          INT     2Ø
```

Machine language

Assembly language

← Assembly —
— Disassembly →

Take heart, however. The mnemonic instructions in the column on the right in the listing, as shown above, are *not* comprehensible to the microprocessor, clever though it may be. They form what is properly called "assembly language," and while you may not understand these instructions now, you soon will even though you are merely a human being.

It is the job of an *assembler* to translate assembly language, which is comprehensible to humans, into machine language, which is comprehensible to microprocessors.

> Assembler programs translate assembly language into machine language.

Assembly-Language Instructions

You've typed in the program, and run it, and disassembled it again, but of course you still don't really understand how it does what it does. To understand the program we must understand the individual instructions in it, and what they do. In this section we'll look at the

instructions one by one. But first we need to understand another fundamental concept: *registers*. So let's digress for a moment, and return to our program later.

Registers

A register is a place in the microprocessor where our program can put a byte, or sometimes two bytes, of data. A register is something like a memory location, except that it has various special properties which a memory location doesn't. One of these properties is that the microprocessor can do a simple kind of arithmetic on the contents of registers; whereas it can only put bytes into, and take them out of, memory locations. However we won't be concerned with this arithmetic capability in this chapter. For the moment, think of registers as places, like memory locations, where we can put eight-bit bytes of data.

The registers in the 8088 are given two-letter names. There are a dozen or so of these registers, but we're going to put off looking at all of them at once until the next chapter. Our particular program concerns itself with only two of the registers: the DL register and the AH register.

There are four instructions in the program, one on each of the four lines in the listing above. The first two deal with registers.

The MOV Instruction

The first instruction, "MOV DL,01", occupies memory locations 100 and 101, and consists of the bytes B2 and 01.

```
Ø8F1:Ø1ØØ B2Ø1          MOV     DL,Ø1
                                  |
                                  Take this number
                                  and MOVe it into
                                  the DL register
```

This instruction tells the 8088, "take the number 01h, and move it (MOV for "move") into the DL register." This way of writing the instruction may seem somewhat backwards to you, moving things from right to left. It's a convention that probably had its origin in the kind of statements used in higher level languages, like BASIC's

```
LET A=2
```

where the quantity on the right gets "assigned" or put into the variable

on the left. At any rate, after this instruction is executed, there will be a byte with the value 1 in the DL register. Where does the 8088 get the 01? It's actually part of the instruction: the second byte. The 8088 microprocessor looks at the first part of the instruction, the B2, in memory, figures out that this means "move the following 8-bit constant into the DL register," and then gets the 8-bit constant from the very next memory location (0101h) and places it in the DL register. The operation of the MOV DL,01 instruction is shown in Figure 2-1.

When we introduce each assembly-language instruction in this book we're generally going to start with a box which summarizes the ways the

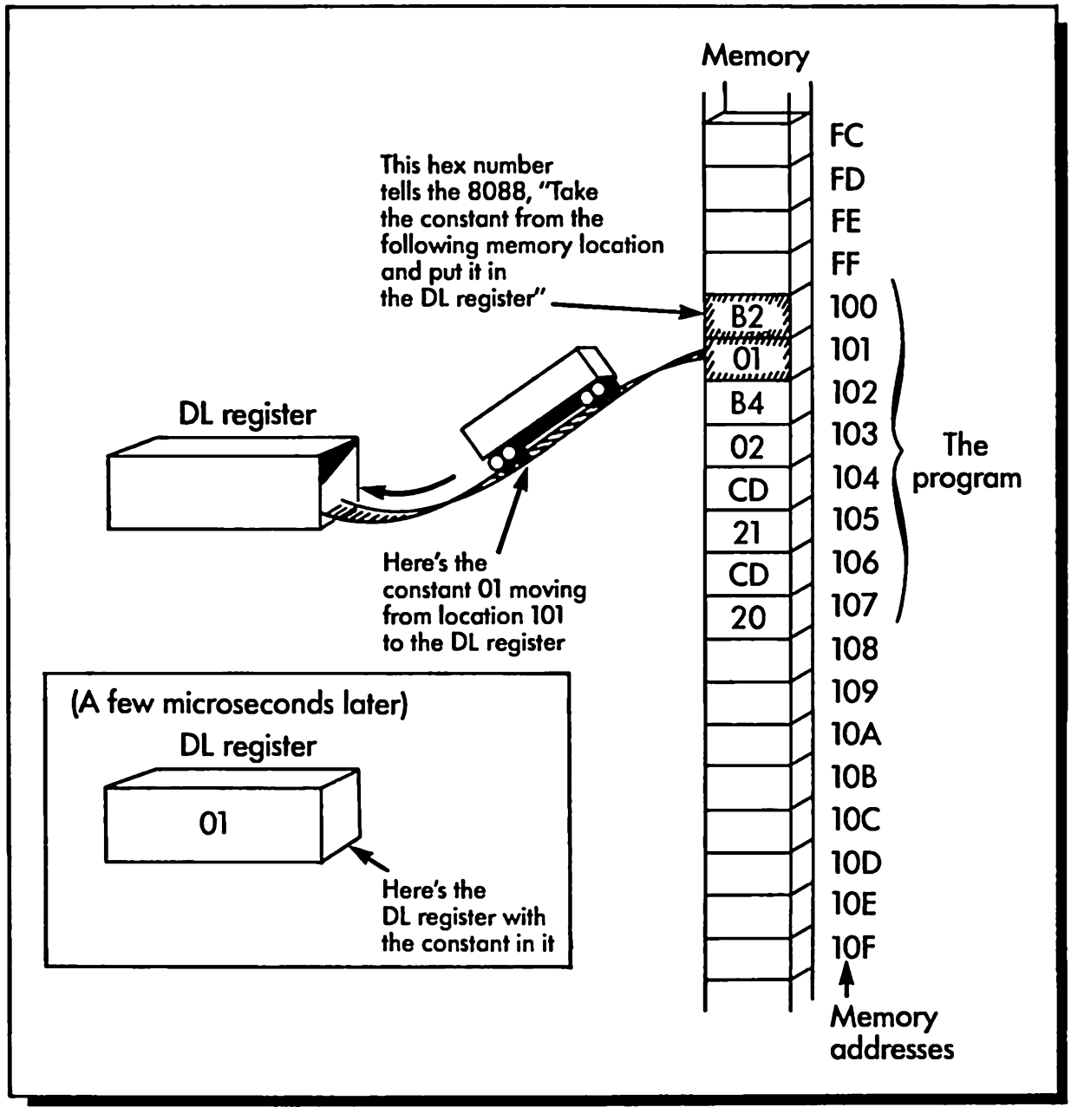


Figure 2-1. Operation of the MOV DL, 01 instruction

instruction can be used; and the MOV instruction — your first assembly language instruction — is no exception. However, you should understand that at this point you don't need to understand everything that's in the box. For our program we're only interested in one use of the MOV instruction: the "immediate to register" byte MOV. As you've seen, this means taking a constant two-digit hex value which is part of the instruction (it "immediately" follows the instruction in memory, hence the name), and putting it in a register. The other uses of this instruction, involving MOVes to and from memory and between registers, will be covered later. Likewise, at a later time we'll also explain the meaning of the word "flags" used in the bottom line.

Thus for the moment you can ignore most of the following box:

MOV Instruction

Moves byte or word from/to register/memory/immediate.

Move immediate value to Register

```
MOV DL,Ø1          ;byte
MOV AX,1234        ;word
```

Move immediate value to Memory

```
MOV MBYTE,12       ;byte
MOV MWORD,1234     ;word
```

Move Register to Register

```
MOV DL,BL          ;byte
MOV AX,BX          ;word
```

Move Register to Memory

```
MOV MBYTE,BL       ;byte
MOV MWORD,DX       ;word
```

Move Memory to Register

```
MOV CH,MBYTE2      ;byte
MOV AX,MWORD4      ;word
```

Flags affected: none

The second instruction in our program is also a MOV instruction:

```
Ø8F1:Ø1Ø2 B4Ø2          MOV     AH,Ø2
```

This means, as you no doubt have figured out, "take the number 02, and MOVe it into the AH register." Because we're moving the constant into a different register, AH instead of DL, the hex code for the instruction is different: B4 instead of B2. And since it's the constant 02h that's being moved into a register, the second byte of the instruction is 02. Otherwise the operation of this instruction is just the same as the first one.

But *why* are we putting these constants in these registers? What does that have to do with printing a happy face on the screen? Fear not, things will become clearer as we describe the next two instructions.

The INT Instruction

INT is a sort of "jump to subroutine" instruction. It stands for "INTerrupt," and there are various reasons why it isn't a real "jump to subroutine," but for the time being we can think of it that way. It's a little like a GOSUB in BASIC or a CALL in various other languages. It transfers control from our program to another routine somewhere else in memory. Then, when that routine is done, control is returned to the line following the INT in our program.

So when the instruction

```
Ø8F1:Ø1Ø4 CD21          INT     21
```

is executed, the program jumps to a special part of DOS, a routine whose number is 21h, and when this routine is finished, control returns to the *next* line of the program, the INT 20 at address 106. This is shown in Figure 2-2.

(Remember, this is a simplified view of INT. Actually the INT instruction involves a transfer to a special address called an "interrupt vector," which in turn transfers control to the routine. However, the effect is much the same as we've shown.

As with the MOV instruction, you can ignore, at least for the moment, a lot of the material in the following box.

INT Instruction

Calls a routine pointed to by an interrupt vector.

Control is transferred with an indirect call through any of the 256 interrupt vectors located from absolute address 00000h to 00400h. The address of the routine, in segment:offset form, must be in the vector.

Control is returned to the calling program from the routine with the IRET instruction.

Flags affected: IF, TF

The Display Output Function

What does this special DOS routine number 21h do? That depends, as we'll see on the number in the AH register at the time we execute the INT 21. Routine 21h is a sort of switchyard, which will route us to a number of different DOS functions, depending on the number in AH. In our case we want to display a character on the screen, so we put the number 2 in the AH register. DOS routine number 21h will then transfer control to the "Display Output" function, whose purpose is to write a single character to the screen.

This Display Output routine is one of the famous "DOS Functions" we've mentioned before. We'll be talking about these functions at length later. For the time being, the important things to know about them are that they are assembly-language routines built into the PC-DOS, so that they are always available in memory when you need them, and that they are all reached by executing an INT 21h instruction, with different values in the AH register.

Your program must do three things to cause the "Display Output" function to actually display a character. First it must put the numeric value of the character to be displayed into the DL register. The numeric value for the happy face is 1. (It's like an ASCII value, except that for special characters like the happy face it's not really ASCII, it's a code IBM invented.) So the first instruction in our program puts 01h into the DL register.

The second thing the Display Output function needs is to have the number 2 put into the AH register, as we explained above. This is the number that tells the operating system that we want the Display Output

function, and not some other function (like Keyboard Input or Print String, which we'll talk about in the next chapter).

The third thing our program has to do is execute the INT 21 instruction itself, so that control will be transferred to the Operating System, which will then look in the AH register to figure out what we want to do, namely, display a character.

As we do with assembly-language instructions, we're going to summarize each DOS function in a box. In the case of DOS functions,

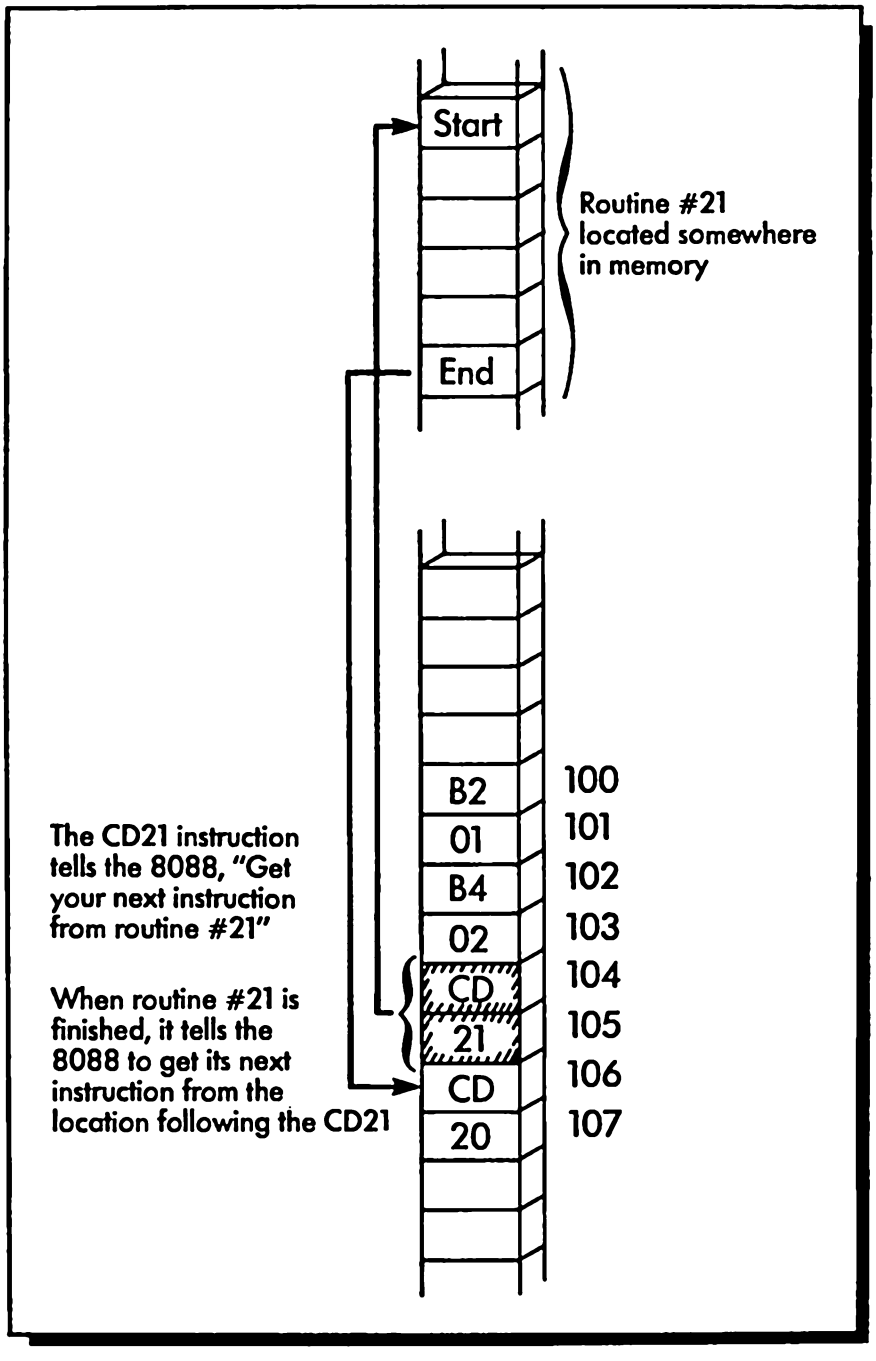


Figure 2-2. Operation of the INT 21 instruction

however, most of the contents of the box should be familiar to you (unlike instructions boxes, which, at this point, leave many unexplained details).

DISPLAY OUTPUT Function — Number 02h

Enter with:

Reg AH = 2

Reg DL = Numeric value of character

Execute:

```
INT 21
```

Return with: character displayed on screen

Comments: Ctrl Break causes exit from function

The instructions that make up the Display Output routine are not in the same place in memory as our program. In fact, we actually don't know where they are, and we don't need to know. The hardware in the 8088 will take care of transferring control from the CD21 instruction in our program, to the beginning of the Operating System, and then getting back to our program when the Operating System has told the Display Output Routine to print the character from the DL register.

Whew — what a lot of complexity in one little instruction. Perhaps Figure 2-3 will help make it clearer.

The Program Terminate Interrupt

The last instruction in the program is another INT instruction, this time to DOS routine number 20h.

```
Ø8F1:Ø1Ø6 CD2Ø          INT     2Ø
```

This routine is much simpler than DOS routine number 21h, in that there's only one thing it can do. Therefore, you don't have to put anything in any of the registers before you call it. The routine is called the "Program Terminate Interrupt." Its job is to ensure that, when a "user program" (such as the one we've just written) has finished executing, it correctly transfers control back to DOS or DEBUG, whichever is being used to run the program (in this case DEBUG).

PROGRAM TERMINATE Interrupt

Execute:

```
INT 2Ø
```

Return with: control returned to supervisor program — DOS or DEBUG

Thus the INT 20 instruction is very similar to a STOP or END instruction in higher-level languages. When the INT 20 instruction has done its work, control goes back to DEBUG and you get the "Program terminated normally" message, and the DEBUG prompt.

Variations on a Theme

Now that we have our program up and running, let's try changing it

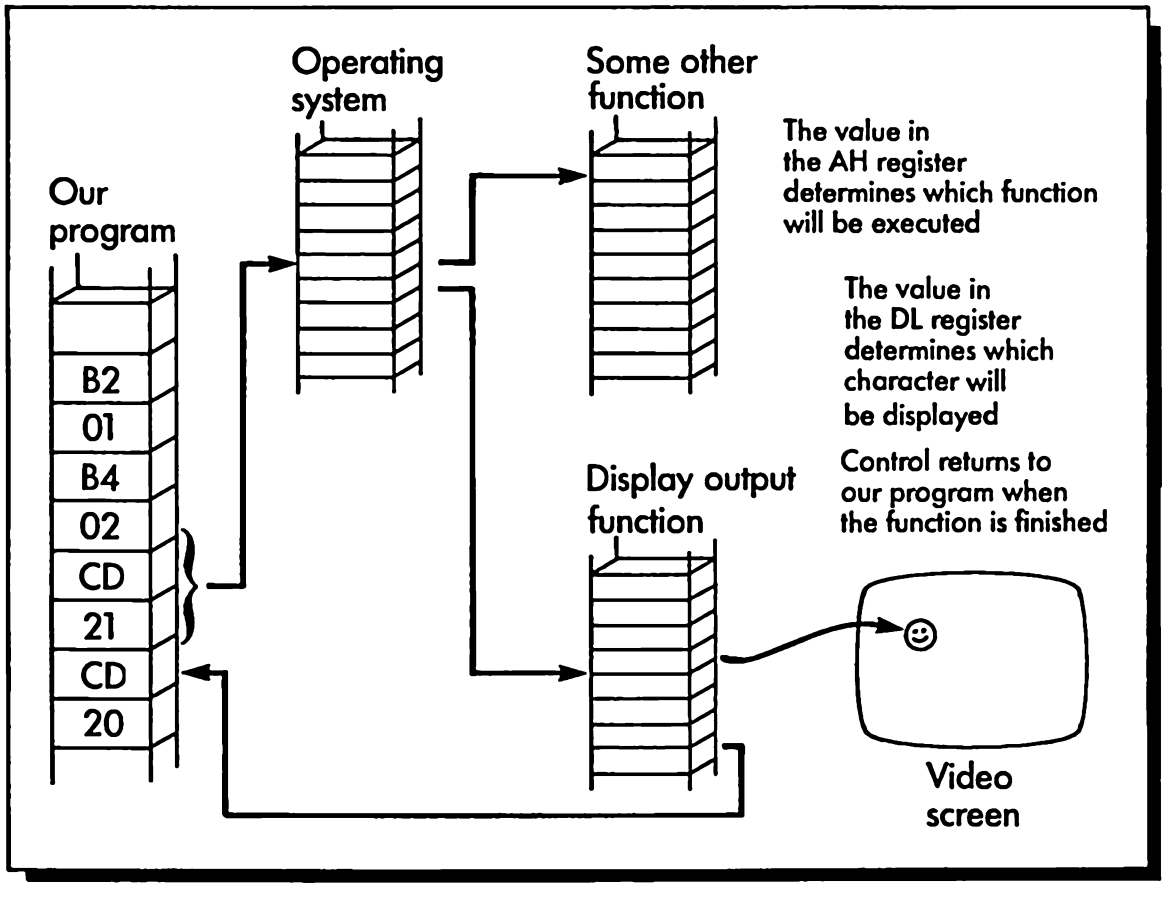


Figure 2-3. The operating system and functions

a little here and there to see what happens. This should give you more understanding of how the program works.

Printing Different Characters

What modifications to this program do you think we would need to make it display some other character, say the letter "X", instead of the happy face? That's right: all we need to do is change the first line, so that instead of MOVing a 1 (the happy face code) into the DL register, we move a 58h, which is the ASCII code for "X".

Assuming that your program is still in memory (if it's not you can type it in again with "A" or "E"), use "U" to look at it again:

```
-u100 107
08F1:0100 B201          MOV     DL,01
08F1:0102 B402          MOV     AH,02
08F1:0104 CD21          INT     21
08F1:0106 CD20          INT     20
```

All we need to do is change *one byte* in this program to make it print an "X". That's the "01" at location 101. We could do this using "A", and simply assemble a new instruction,

```
mov dl,58
```

right over the top of the old one at location 101. But since we only need to change one byte, let's use the "E" command instead.

We want to change the byte at location 101 from 01 to 58h, so we enter "E" followed by that address:

```
-e101                   <-- Enter "e" and the address
08F1:0101  01.58        <-- Type "58"
-
```

Press ↵ here

Old value

Since we only want to put one byte into memory, we press ↵ immediately following the byte.

So we've changed the 01 to a 58h. If we list the program again with "U" we'll see the change incorporated in it:

```
-u100,107
08F1:0100 B258          MOV     DL,58
08F1:0102 B402          MOV     AH,02
08F1:0104 CD21          INT     21
08F1:0106 CD20          INT     20
```

Now if we run it again with "G" we should see an "X" displayed on the screen:

```
-g
X
Program terminated normally
```

Not bad! It worked again. By looking up the ASCII values of various characters, you can change the program to print whichever one you want.

The Endless Loop

Before we go on to a more thorough discussion of assembly language, let's do one more variation on this program. Suppose instead of printing *one* character, we wanted to print a whole *series* of the same character. How would we modify the program to do that? It's not hard: we simply put a "jump" instruction at the end of the program, which takes us back to the beginning so that our "Display Output" function will be repeated over and over.

Let's also go back to the happy face — it's more upbeat than the "X". Put the number 01 back into location 101

```
-e1Ø1
Ø8F1:Ø1Ø1  58.1
```

to restore the happy face.

Now we'll install a new instruction in our program. This instruction goes at the end of the program, overlaying the INT 20 instruction at location 106. It's a "jump" to the beginning of the program, at location 100, so the program becomes an endless loop.

If you're using DOS version 2, enter "a106", and when the address is printed, enter "jmp 100". Then on the next line hit ↵ again to go back to DEBUG.

```
-a1Ø6
Ø8F1:Ø1Ø6 jmp 1ØØ      ← Enter jmp 100
Ø8F1:Ø1Ø8               ← Press ↵
```

If you're running DOS version 1, you'll have to type in the hex code for this instruction using "E". The hex code is EBF8h.

```
                Old values
-e1Ø6          /       \
Ø4B5:Ø1Ø6  CD.EB    2Ø.F8
               \       /
                New values
```

Now "unassemble" the program to make sure it looks right:

```
-u1ØØ,1Ø6
Ø8F1:Ø1ØØ B2Ø1          MOV     DL,Ø1
Ø8F1:Ø1Ø2 B4Ø2          MOV     AH,Ø2
Ø8F1:Ø1Ø4 CD21          INT     21
Ø8F1:Ø1Ø6 EBF8          JMP     Ø1ØØ
```

The JMP Instruction

The box containing the summary of the JMP instruction is largely for later reference. You can ignore most of it at this point.

JMP Instruction

Jumps to new memory location.

Within-segment short jump: to address within −128 to + 127 bytes

```
JMP NEAR_LABEL
```

Within-segment long jump: to address in same segment

```
JMP NEAR_LABEL
```

Intersegment jump: to address in a different segment

```
JMP FAR_LABEL
```

The last two types can also be "indirect jumps," that is, jumps to the memory address contained in a memory address, a register, or a memory address modified by a register.

```
JMP WOR_VAR
JMP AX
JMP ADDR_PTR [BX]
```

Let's look at the JMP instruction a little more closely, to see how "JMP 100" gets assembled into "EBF8." You don't need to remember the details of this process, but it will give you some idea of what DEBUG's "A" command (or the assembler) has to figure out to arrive at the correct machine language equivalent of a particular assembly-language instruction.

The first two hex digits that make up the instruction are EB, which is the code for a "short" jump. (We'll talk about the difference between long and short jumps later.) What does the F8 mean? You might expect to see the address 100 that we're jumping to, but you don't. This is because the jump is a *relative* jump. Instead of using the *address* of the place we're going to jump to, JMP uses the *distance to* the place we're going to jump.

Even after you know this, the F8 still doesn't make much sense. There are two reasons for this. The first is that since the jump is *backwards*, the number of bytes that need to be jumped is *negative*. If we were going to jump forward eight bytes, the instruction would simply be EB08. But since we're going to jump backward eight bytes, we form a negative number by subtracting 8 from 00. If you had FF and added 1 to it, you'd get 00. So if you have 00 and you subtract 1 from it, you get FF. Subtract 1 again and you get FE. Count down by 1, eight times, and you get FD, FC, FB, FA, F9, F8.

But what have we jumped 8 bytes *from*? This brings us to the second reason the F8 is confusing: it doesn't tell you to jump 8 bytes from the jump instruction itself, it tells you to jump 8 bytes from the byte *following* the jump instruction. That would be from location 108 to location 100, which is in fact 8 bytes. Expressed arithmetically, this looks like

```
1ØØh - 1Ø8h = F8h
```

Whew. What a lot of complexity in just one little instruction. Fortunately the "A" command (and, as we'll see later, the assembler programs ASM or MASM) do all these tedious calculations for us, so that we don't even have to think about the hex representations of instructions unless something goes wrong.

DOS version 1 users are at a disadvantage here, since they have no easy way to start with the assembly language mnemonics and end up with the machine language numbers, at least not using DEBUG. Thus if version 1 users were trying to figure out the hex equivalent of the JMP 100 instruction so they could type them in, they'd either need to go through the calculation just described, or else try to figure it out by trial and error, guessing a number and then using "U" to see if it was right.

Actually version 1 users don't need to do either of these things for the

examples in this book, since we supply all the hex equivalents, so that "E" can be used immediately. You can find these hex codes by looking at the "U" listing included with each program.

In and Out of the Endless Loop

Have you waited all this time to try the new program? Good — that shows admirable restraint. Try it now. Enter the "G" command:

```
-g
```

Wow — look how fast the screen fills up with happy faces! All right, now stop the program. How do you do that? Just hit Ctrl Break (the Ctrl and Break keys together) as you would for any other program, and presto, we're back in DEBUG. This is possible because the Display Output function is programmed to look for Ctrl Break at the same time it's printing characters on the screen. We can thus tell it, "Stop! Don't print that character — I want to get back to DEBUG!"

When you escape from your program back to DEBUG you'll get a display like this:

```
AX=Ø2Ø1  BX=ØØØØ  CX=ØØØØ  DX=ØØØ1  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1Ø6   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1Ø6 EBF8          JMP     Ø1ØØ
```

Don't worry about what all this means at this point — we'll look into it later. Do note, however, that the last line of the display contains one of the instructions from your program, in this case the JMP 0100. This shows what instruction was being executed when you pressed the Ctrl Break keys.

Perhaps this is a good time to end the chapter, with the screen (mostly) full of happy faces.

Summary

In this chapter you've learned a good bit about DEBUG, written your first assembly-language program and some variations on it, and explored some of the ways to see what your computer is doing on a very fundamental level.

We hope that this chapter has given you an idea (although at this point it will be a somewhat impressionistic idea) of what assembly language is all about, and whetted your appetite for a more detailed understanding.

3

What Is Assembly Language?

Concepts

Machine language versus assembly language
Registers
Saving programs to disk from DEBUG, and loading them back
Input/Output ports
Logic instructions
Toggling a bit to beep the speaker

Debug Commands

R = Registers
N = Name
W = Write
Q = Quit
L = Load

8088 Instructions

INC = Increment
LOOP = Loop
IN = Input
OUT = Output
AND = Logical "AND"
XOR = Logical "Exclusive OR"

Applications

SMASCII program — Displays entire character set
SOUND program — Beeps the speaker

In the last chapter we led you straight into the heart of the 8088 microprocessor. We showed you how to examine memory, write

programs, assemble them, disassemble them, and execute them. We did this to show you that programming in assembly language doesn't have to be all that difficult. However, there were some details that we left out along the way.

In the first part of this chapter we're going to fill in some of these details. Then we'll go on to write some more programs, to consolidate what we've learned.

Filling in Details

An assembly-language programmer is primarily concerned with three things: instructions, memory, and registers. In the process of writing our first assembly-language programs in the last chapter we talked a little about each of these topics. In the following sections we'll take a somewhat more leisurely look at each of the three, and try to deepen your understanding of what assembly language is all about.

Machine Language, Assembly Language, and Physical Reality

Although we've talked so far about assembly-language instructions, and the data they operate on, in terms of hexadecimal numbers, the fact is that if we look at things in a more fundamental way we should be talking about *binary* numbers, not hexadecimal numbers. Let's look at an example of what we mean.

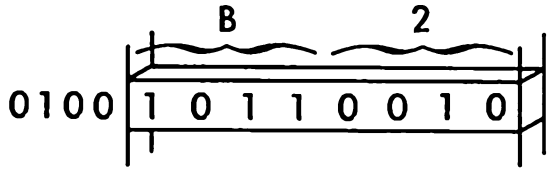
Remember the happy face program you wrote in the last chapter? It looked like this when you disassembled it with "U":

```
-u100,107
08F1:0100 B201          MOV     DL,01
08F1:0102 B402          MOV     AH,02
08F1:0104 CD21          INT     21
08F1:0106 CD20          INT     20
```

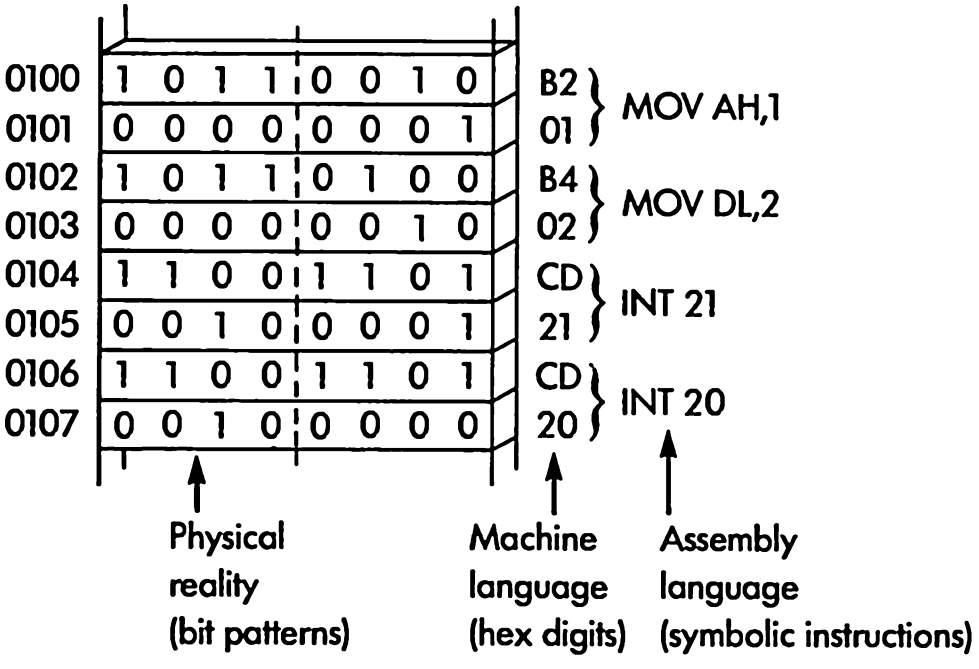
As you learned, the symbolic statements on the right of this listing constitute a form of *assembly language.* The mini-assembler in DEBUG translates these statements into the hex numbers in the columns on the left. These hex numbers are called *machine language.*

Although we glossed over this point in the last chapter, it is not actually the hex digits themselves which are read and understood by the 8088 microprocessor, but the binary numbers or *bit-patterns* they represent. Hex digits are merely a way to make binary digits easier for us humans to read.

For instance, the instruction MOV DL,01 in the program above is translated into the hex number B201. The B2 part of this instruction goes in location 100. However, B2 is not stored in the computer's memory as a hex number, but as a pattern of bits:



In fact, the entire program is stored in memory as a pattern of bits:



Notice how each of the instructions in our program occupies a specific place in the computer's memory. In this program all the instructions are two bytes long, but other instructions can vary in length from one byte to five bytes, and sometimes even more.

The earliest microcomputers (such as the venerable IMSAI-8080) had lights on the front panel which could be set to show the bit patterns inside the computer. It was possible to "step through" a program and look at the binary representations of all the program instructions, much as we've shown you in the diagram above. This was occasionally helpful in debugging some complex internal process (especially for hardware-oriented users).

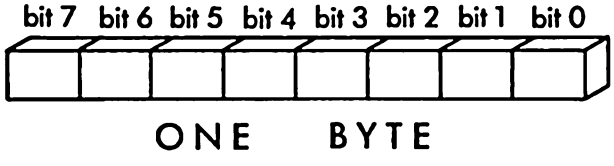
If we had such an old-fashioned computer with front panel switches, we could actually look at, say, memory location 0100 and see the lights lit where the bits were turned on, as shown in Figure 3-1.

Since the PC is so modern, it doesn't have these lights, but DEBUG

shows us the contents of memory just as well — in fact better, because DEBUG is far more convenient to use than a bunch of front panel switches, and also because it's far easier to read four hex digits than a line of sixteen lights. However, using DEBUG does deprive us of a certain insight into what happens deep inside the computer: it's easy to forget that the hex digits which DEBUG shows us are not really what's in the computer's memory and registers: these hex digits merely stand for *a binary pattern of bits.* (Later in this chapter we'll see how important actual bit patterns can be: we'll write a program which must manipulate bits in order to make sounds on the speaker.)

Bit Numbering

What is memory? From an assembly-language programmer's viewpoint, memory is a place in your computer where you can store something called "bytes." A *byte* is simply eight bits arranged in a row. A *bit* is the smallest possible unit of information: either yes or no, on or off, 1 or 0. Thus each memory location consists of eight places where bits can be stored, like this:



The way the bits are numbered — whether from right to left or left to right — is largely arbitrary. The figure above shows how IBM likes to do it, as do most computer manufacturers, but some manufacturers start with 0 (or sometimes 1) on the left instead of the right. Each bit can have a value of either 0 or 1, so if we placed the byte represented by, say, the

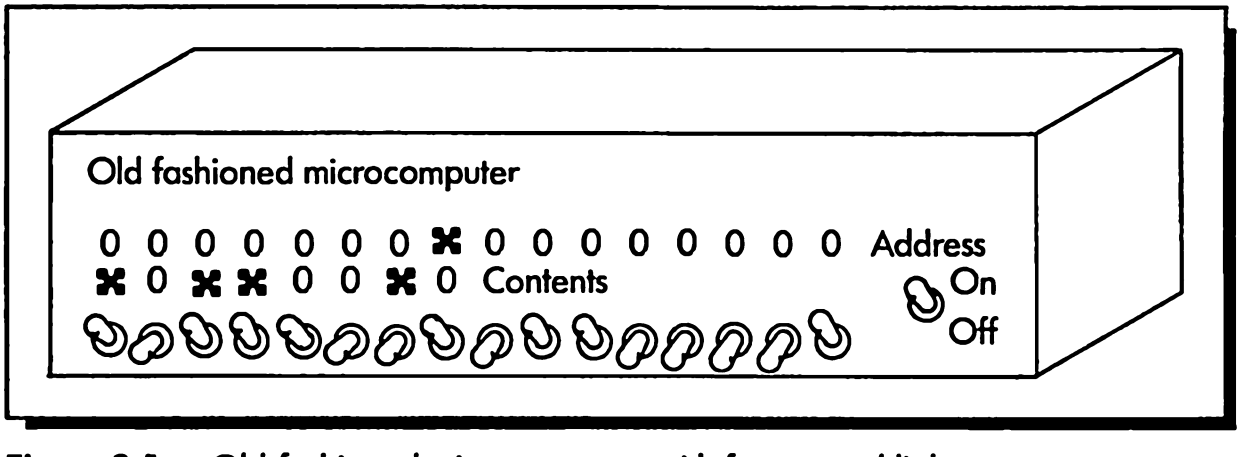
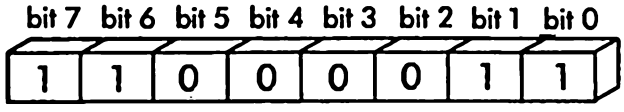
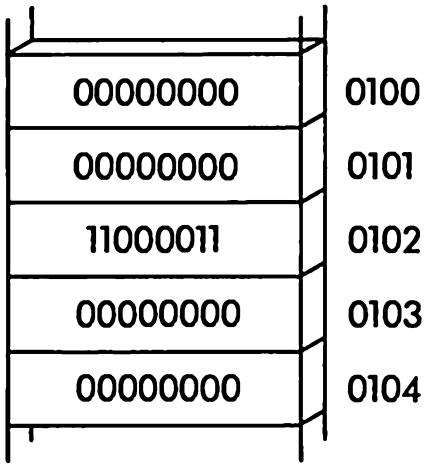


Figure 3-1. Old-fashioned microcomputer with front-panel lights

hex number C3 (which is 11000011 in binary, since C=1100 and 3=0011) into a memory location, the location would look like this:



In a computer's memory there are many thousands of locations like this, into which 8-bit numbers can be placed. Here's how a small section of memory looks, filled in with binary numbers:

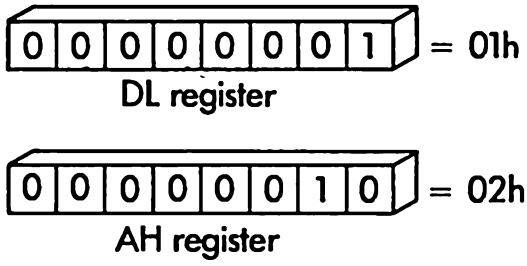


There's our friend, C3, in location 102. The other locations happen to contain zeros.

As we discussed in the last chapter, we are, for the next few chapters, confining ourselves to a single segment of memory, or 65536 bytes. Although this is only a fraction of the memory the IBM PC can hold, it is nevertheless a large number of bytes. When we show figures, such as the one above, which depict a half-dozen bytes of memory, it's important to remember that in reality the memory locations in our segment start at 0000 and go all the way up to FFFFh or 65536d, as shown — somewhat fancifully — in Figure 3-2.

Registers

In the last chapter you were introduced to two registers: AH and DL. We used the AH register to hold a number that told the operating system what DOS function we wanted to perform when we executed an INT 21 call to DOS, and we used the DL register to hold the numerical ASCII value of a particular character to be displayed. We showed these 8-bit registers as containing pairs of hex numbers, but of course what each really contains is eight bits:



As we mentioned, a register is a physical device built into the computer. It's something like an address in memory, but since it's part of

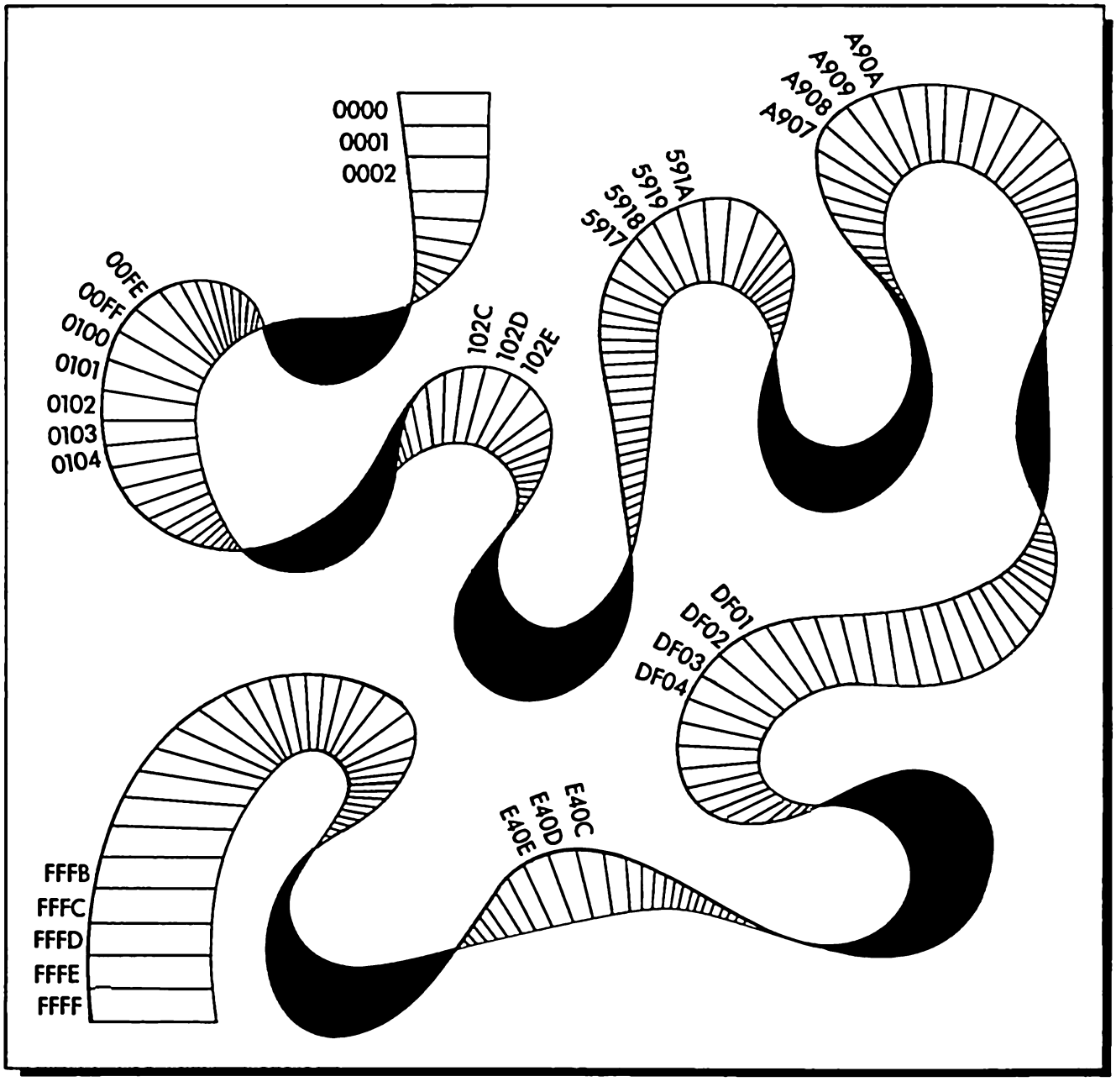


Figure 3-2. One segment of memory

the microcomputer chip itself, rather than part of a memory chip located somewhere else on the computer, data can be moved from register to register very quickly. The 8088 instructions can also do a far wider variety of things to registers than they can to memory locations. For instance, arithmetic and logical operations can be performed on data in registers, addresses stored in registers can be used to point to locations in memory, and registers can be used to read and write data to the peripherals connected to the computer.

Although so far we've shown you only two registers, there are actually *eight* 8-bit general-purpose registers. (There are some other more specialized registers as well, but we'll ignore them for the time being.) These eight registers are called:

AH and AL

BH and BL

CH and CL

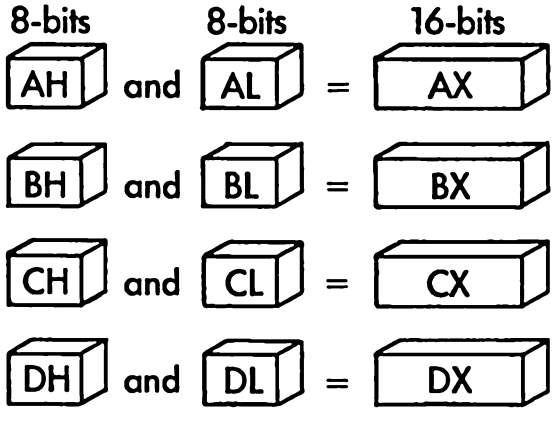
DH and DL

As you can see, we've arranged the eight registers in pairs. That's because they're arranged in pairs in the 8088 microprocessor. Why is this?

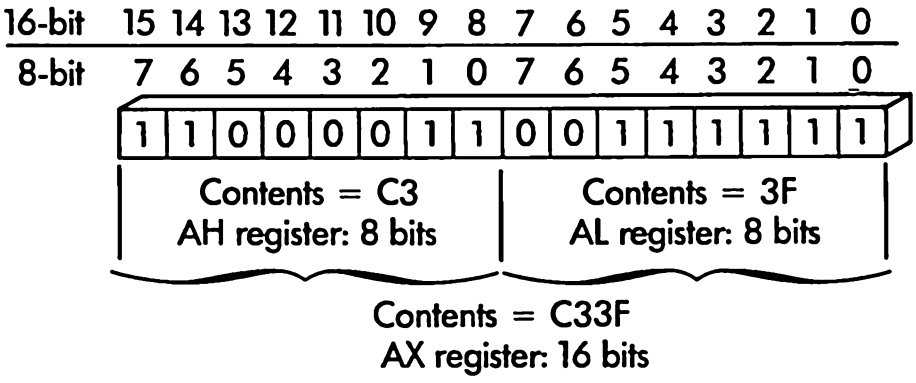
Some data which we want to manipulate is 8 bits wide, as we saw in the programs in the last chapter. However, we often want to be able to operate on 16-bit-wide data. This data might be just numbers, or it might be addresses: as we've seen, a 16-bit number can specify any address in our current 64K data segment.

Instead of having some 8-bit registers and other 16-bit registers, the designers of the 8088 decided to group pairs of 8-bit registers together to form 16-bit registers.

In 8088 assembly-language format, the 16-bit register is differentiated from the two 8-bit registers by giving it the same first letter as the pair from which it was made, but ending it in the letter "X":



A more detailed picture, showing the positions occupied by individual bits in the AX register, looks like this:



The upper row of bit numbers shows how the bit positions are numbered in the 16-bit register, while the lower row shows the numbering for the two halves of the register used as two separate 8-bit registers. The "H" in the register name stands for "high," and the "L" stands for "low," since (for instance) the AH register forms the high part of the AX register, containing the most significant bits, and the AL register forms the low part, with the least significant bits. For example, the most significant digits of the 4-digit hex number C33F are C3, shown in the AH register in the figure above, and the least significant digits are 3F, in the AL register.

These four 16-bit registers, AX, BX, CX, and DX, are in many ways identical to one another. However, depending on the circumstance, they may have areas of specialization. The AX register has special circuitry that makes it more suitable for doing arithmetic and logical operations than the other registers. The BX register can be used to point to memory addresses in a way that other registers can't, and the CX register is often used for counting. As we learn more about assembly-language instructions we'll begin to see how these specialized features of the different registers are used.

Manipulating Registers with DEBUG

DEBUG has a command which enables us not only to look at the registers in the 8088 (really at the hex representations of their contents), but to change the contents as well.

Load DEBUG as described in the last chapter, and when the prompt appears, type "R" (for "Registers"). You'll be rewarded with a very complicated looking display:

```
A>debug
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1ØØ ØØØØ          ADD     [BX+SI],AL                      DS:ØØØØ=CD
```

This is the same kind of display you saw in the last chapter when you terminated the endless loop program with Ctrl Break.

For the time being, you can ignore most of the display. Notice, however, the first four entries on the top row:

```
AX=ØØØØ  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ
```

This tells you that the contents of all four major registers in your 8088 are set to zero. The contents of the registers will change when a program executes instructions that put data into the registers, as your programs did in the last chapter with MOV DL,01 and MOV AH,02. For this reason, if you use the "R" command after running a program, you may find that some of the registers contain values other than zero.

There is another way to change the contents of the registers: You can do it directly from DEBUG, using a variation of the "R" command. For instance, enter the command "R" followed by the letters "AX" (for the AX register). DEBUG will respond by printing out the letters "AX", followed by the current contents of AX, which in this case is 0000. It then prints a colon and sits there waiting for you to enter a number representing the new contents of AX.

```
-rax
AX ØØØØ
:              ←— DEBUG waits for you to enter a new value
```

Suppose you now enter "1234".

```
-rax
AX ØØØØ
:1234          ←— You enter 1234
-              ←— Back to the DEBUG prompt
```

This places the hex number 1234 into the AX register. This is the same as putting 12 in the AH register, and 34 in the AL register. With DEBUG, you can't access the two halves of the register separately — you must deal with them both together. To verify that what you put in AX is really there, type just plain "R" again:

```
-r
AX=1234  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1ØØ ØØØØ           ADD     [BX+SI],AL                          DS:ØØØØ=CD
```

And there it is, 1234 in the AX register.

Similarly, let's put FFFFh into the CX register:

```
-rcx          <— Enter this for "Register CX"
CX ØØØØ       <— Current contents is ØØØØ
:ffff         <— Change it to FFFFh
-
```

Checking again with "R", we get:

```
-r
AX=1234  BX=ØØØØ  CX=FFFF  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1ØØ ØØØØ           ADD     [BX+SI],AL                          DS:ØØØØ=CD
```

You can do this with any of the four major registers: AX, BX, CX, and DX; and with a variety of other registers as well, although we're going to ignore these other registers for the time being.

This ability to examine and modify the contents of registers directly from DEBUG will become important when we learn how to follow the operation of your program while it's running, a topic we'll cover later when we talk about the "T" (for "Trace") command.

ASCII Display Program

Let's write another program. This one will display all the ASCII characters (and all the special non-ASCII IBM characters as well) on the screen. It will also introduce you to a new instruction. Once you've written the program, we'll show you how to save it on your disk, so that you can execute it directly from DOS without getting into DEBUG.

From DEBUG, use the "A" command to type in the following little program. As you can see, we've added comments to each line. You can't type these in, since DEBUG doesn't accept comments, but when we explain the program, they'll help clarify its operation.

```
A>debug
-a
Ø8F1:Ø1ØØ mov dl,Ø          <— Put first character in DL
Ø8F1:Ø1Ø2 mov ah,2          <— Specify Display Output function
Ø8F1:Ø1Ø4 int 21            <— Call DOS to print character
```

```
Ø8F1:Ø1Ø6 inc dl          ←— Change to next character
Ø8F1:Ø1Ø8 jmp 1Ø2         ←— Go back to display next character
Ø8F1:Ø1ØA                 ←— Press [↵] to end assembly
```

Note that the first time you use "A" after calling DEBUG, you don't need to specify "a100". DEBUG *assumes* you want to start at 100h unless you tell it otherwise. Use "U" to see that everything looks all right:

```
-u1ØØ,1Ø8
Ø8F1:Ø1ØØ B2ØØ          MOV     DL,ØØ
Ø8F1:Ø1Ø2 B4Ø2          MOV     AH,Ø2
Ø8F1:Ø1Ø4 CD21          INT     21
Ø8F1:Ø1Ø6 FEC2          INC     DL
Ø8F1:Ø1Ø8 EBF8          JMP     Ø1Ø2
```

(If you're using DOS version 1, you'll have to use "E" to type in the hex numbers B2, 00, B4, 02, CD, 21, FE, C2, EB, F8, as we explained in the last chapter.)

The INC Instruction

As you can see, this program uses a new instruction which you haven't seen before: INC. The purpose of this instruction is to *increment* — that is, add one to — the contents of a register.

INC Instruction

Increments the contents of a register or memory address.

To increment a register:

```
INC  BX
INC  AL
```

To increment a memory address:

```
INC  WORD_VAR
INC  BYTE_VAR
INC  TABLE [BX]
```

Flags affected: AF, OF, PF, SF, ZF

Figure 3-3 shows how the INC DL instruction works.

Operation of the ASCII Program

What does this INC instruction do in our program? As you can see, the first three instructions of the program are very similar to the program in the last chapter which printed a happy face on the screen. The only difference is in the first instruction: the constant loaded into the DL register to be printed is 0 instead of 1 (1 is the code for a happy face). So we can surmise that the first thing the program is going to do is print something: whatever the character is that corresponds to 0.

The JMP instruction at the end of the program should also be a familiar sight. From the last program in the last chapter, we know that this JMP takes us back to the start of the program, turning the program into an endless loop.

That leaves only the INC instruction unexplained. Its purpose is to increment — add one to — the DL register, every time we cycle through the program. Since the value in the DL register determines what character will be displayed when we call the Display Output function, we can see that a different character will be displayed each time. The first time through the loop this number will be 0, then 1, and so on up to FFh, which is 255d. This is as high a number as can be expressed in a single byte, so the next time through the loop, DL will be back at 0. Each of the numbers from 0 to FFh represents a different character, so the program will show them all to us on the screen, over and over again.

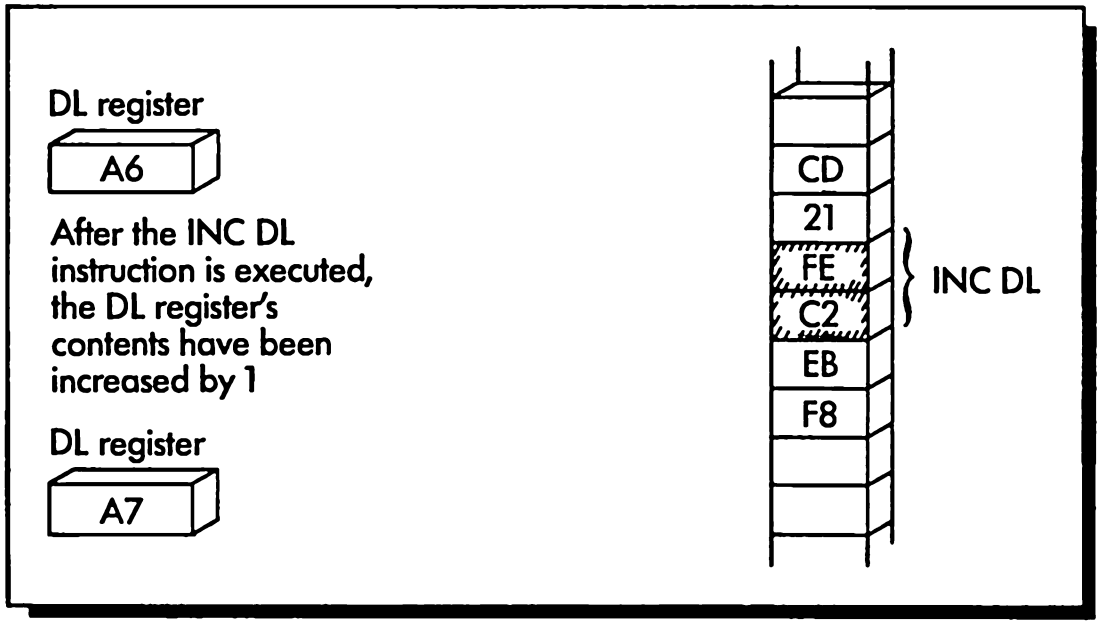


Figure 3-3. Operation of the INC DL instruction

Saving the Program to Disk

Before you run the program, save it on your disk. This is a three-step process: first we tell DEBUG the name of the program we want to save, then we tell it how large the program is, and finally we tell it to actually write the program to the disk.

To specify the name, we type the command "N" (for "Name"), followed immediately (no space) by the name of the file we want the program saved under. The *filename* itself can be anything we want, but the *extension* must be COM if we want to execute the program later directly from DOS. This is because a COM file has certain attributes that make it compatible with DEBUG. (We'll have more to say about this later, in chapter 8, on memory segmentation.) We'll call our program ASCII, so we enter:

```
-nascii.com
```

Next, to tell DEBUG how long a program is (that is, how many bytes we want to save), we use both the BX and the CX registers. The CX register holds the low-order, least significant part of this number, and the BX register holds the high-order part. Notice that we're now talking about 16-bit wide registers, which can hold numbers up to FFFFh, or 65536d. A program this long is a big program. Since by using only the CX register we can save programs up to this length, the chances are we'll never have to put anything in the BX register. However, we must be sure it's set to zero. Then we put the actual number of bytes occupied by our program into the CX register.

You need to be a little careful figuring out this number. If the program starts at 100 and the last byte used is 109 (as in the program above), then how many bytes must be saved? Not 9, but 10d, which is Ah. This is because we start counting at 100, not 101. So we enter:

```
-rbx          <— Enter this to see the BX register
BX ØØØØ       <— It's already set to zero
:             <— Just press ↵ here, since it's already zero
-rcx          <— Enter this to see the CX register
CX ØØØØ       <— It's zero too
:a            <— Enter number of bytes in program
```

Finally, to actually write the program, we enter "W" (which stands for guess what).

```
-w                      <— You type this
Writing ØØØA bytes      <— DEBUG tells you this
```

DEBUG then tells you how many bytes it's writing, and that's it — the program is on the disk. (If you had wanted it on a different drive you could have entered, for instance, –nb:ascii.com. when you named it.)

The "Q" Command

Now you can execute the program directly from DOS. To do this you need to exit from DEBUG to the operating system. There's a DEBUG command for this: "Q" for "Quit." Enter it, and presto, you're back in PC-DOS.

```
-q      <— Enter this to escape from DEBUG

A>      <— You're back in DOS
```

Now that you're back in DOS you can see if the program has in fact been saved on the disk. Do it in the usual way, with the DIR command. If it's not on the disk, you probably made a typing mistake somewhere in the "save" process.

Executing the Program from DOS

Executing COM programs from DOS is easy: you just type the name of the program.

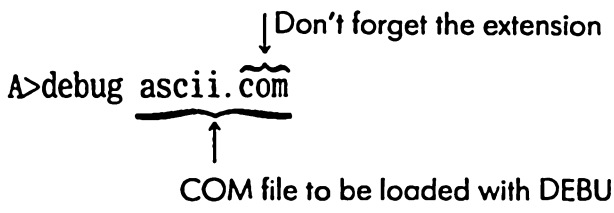
```
A>ascii
```

Wooey, look at all those characters fill up the screen! The beeper, which is also a character (7h), sounds too. To stop the program, hit Ctrl Break.

Notice that you have now created a complete, stand-alone, run-it-anytime program. If someone wants to see the character set available on your IBM, all you have to do is power up the system, wait for the A> prompt, and enter the program name "ascii". If they want a copy of the program, you can copy it to another disk just as you would any other file or program. In other words, you've written a perfectly good assembly-language application program! It may not be quite as useful as a word processor or a spreadsheet program, but the concept is the same.

Reloading the Program in DEBUG

If you want to modify or examine the program again, there are two ways to get it back into DEBUG. The first way is to enter the program name at the same time you load DEBUG. The thing to remember here is that you must use the full filename, *including the extension,* as shown here:



You can also load DEBUG first, and *then* load the program. This is done by first filling in the program name with the "N" command, and then loading it with the "L" (for "load") command. "L" is the opposite of "W" —it causes the COM file to be loaded back from the disk into DEBUG. Here's how that looks:

```
A>debug
-nascii.com
-l
```

You'll hear the disk drive whirr, and if you use "U", you'll see that your program is back in memory again.

```
-u1ØØ,1Ø8
Ø8F1:Ø1ØØ B2ØØ          MOV     DL,ØØ
Ø8F1:Ø1Ø2 B4Ø2          MOV     AH,Ø2
Ø8F1:Ø1Ø4 CD21          INT     21
Ø8F1:Ø1Ø6 FEC2          INC     DL
Ø8F1:Ø1Ø8 EBF8          JMP     Ø1Ø2
```

Now if you want you can also run it directly from DEBUG by using the "G" command, just as you have done with the other programs we've written. The moral here is that as far as DEBUG is concerned, a program can either be typed in with "A" or "E", or it can be loaded in from the disk with "L": the result is the same. Try running the program:

```
-g      ←— Enter "G" to run the ASCII program from DEBUG
```

Again, the screen will fill up with the character set, over and over again.

SMASCII — Making the ASCII Program Smarter

It would be somewhat more elegant if our ASCII program only printed the character set once, and then returned to DOS (or DEBUG, if we ran it from there) without our having to interrupt it with the Ctrl Break keys. Let's modify it to do that, and at the same time introduce another 8088 instruction: LOOP.

From DEBUG, type in the following program (not the comments, of course):

```
-a1ØØ
Ø9Ø5:Ø1ØØ mov cx,1ØØ        ←— Set up the count for the LOOP
Ø9Ø5:Ø1Ø3 mov dl,Ø          ←— Put the first character in DL
Ø9Ø5:Ø1Ø5 mov ah,2          ←— Specify Display Output
Ø9Ø5:Ø1Ø7 int 21            ←— Call DOS to display character
Ø9Ø5:Ø1Ø9 inc dl            ←— Change to next character
Ø9Ø5:Ø1ØB loop 1Ø5          ←— Loop until CX is zero
Ø9Ø5:Ø1ØD int 2Ø            ←— Exit to DEBUG or DOS
Ø9Ø5:Ø1ØF
```

Here it is disassembled with "U":

```
-u1ØØ,1Ød
Ø8F1:Ø1ØØ B9ØØØ1        MOV     CX,Ø1ØØ
Ø8F1:Ø1Ø3 B2ØØ          MOV     DL,ØØ
Ø8F1:Ø1Ø5 B4Ø2          MOV     AH,Ø2
Ø8F1:Ø1Ø7 CD21          INT     21
Ø8F1:Ø1Ø9 FEC2          INC     DL
Ø8F1:Ø1ØB E2F8          LOOP    Ø1Ø5
Ø8F1:Ø1ØD CD2Ø          INT     2Ø
```

This program is very much the same as the ASCII program shown earlier, except that it starts off with a MOV CX,100 instruction, and finishes with LOOP 105 and INT 20. We've already learned that the INT 20 is a sort of "exit" instruction. What about MOV CX,100 and LOOP 105?

The LOOP Instruction

LOOP is a powerful instruction that functions somewhat like a FOR...NEXT loop in BASIC. The idea is this: You put a number, which is the number of times you want to do something, into the CX register. Then, every time you execute the LOOP instruction, it *decrements* (that is, subtracts 1 from) the contents of the CX register. LOOP then jumps back to the address written as part of the LOOP instruction, in this case, 105. More accurately, it jumps to this address *unless the count in CX is zero.* If CX contains zero, no jump takes place, and the program goes on to the instruction following the LOOP. In other words, when CX goes from 1 to 0, LOOP stops looping. Figure 3-4 shows the operation of the LOOP 105 instruction.

LOOP Instruction

Jumps to start of loop until CX register is zero.

The number of times the loop is to be executed must be placed in the CX register before LOOP is invoked:

```
        MOV   CX, COUNT
START:
```

(instructions within loop)

```
    LOOP START
```

Flags affected: none

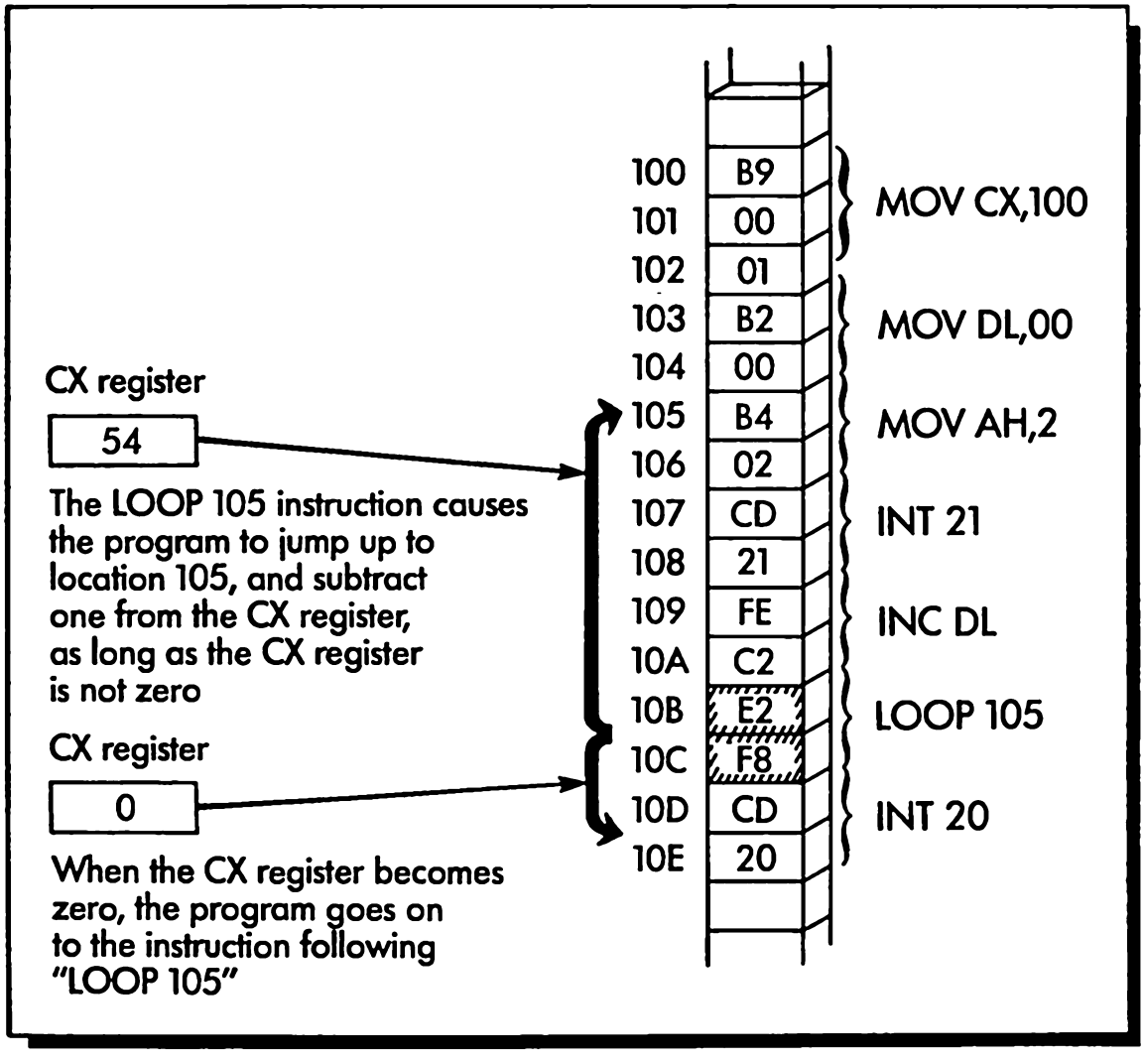


Figure 3-4. Operation of the LOOP 105 instruction

The effect is that all the instructions between the LOOP and the address pointed to by LOOP will be executed the number of times corresponding to the value originally placed in the CX register. In our case we want to print out all 256d possible characters, from 0 to 255. The hex equivalent of 256d is 100h, so we put 100h in the CX register before we begin our loop. The program will then keep incrementing the number in the DL register from 0 to 255d, just as it did in the last program. However, when the count in CX reaches 0, the instruction following the LOOP will be executed, and control will pass to the INT 20 instruction, which will terminate the program.

Before you try out the program, save it to your disk. You can go ahead and execute it first with "G" if you want, but saving a newly written program to the disk before you run it is a good habit. That way if there's a bug in the program — or a design defect — that causes DEBUG and/or the entire operating system to crash, you won't lose the program.

```
-nsmascii.com          <— Set the program name
-rbx
BX 0000
:
-rcx                   <— Set the length to Fh
CX 0000
:f
-w                     <— Write it to the disk
Writing 000F bytes
```

Now you can execute the program from DEBUG with "G", or you can get out of DEBUG and execute the program directly from DOS:

```
-q

A>smascii
```

If you wrote it right, it'll put the entire IBM character set on the screen, once; then return you to the DOS prompt. This is a nice refinement over ASCII, which went on and on (until you hit Ctrl Break).

Some Sound Advice

Before we wrap up this chapter, we're going to introduce you to one more feature of your PC: its ability to produce sound. You have no doubt seen the little speaker grill in the front of your machine, and heard it go

"beep" on start-up, and when, for instance, you type in too many characters for the keyboard buffer to hold. You may also have used the built-in BEEP, SOUND, and PLAY statements in BASIC. In this section we're going to show you how to control this sound capability with assembly language.

Assembly language gives you far better control over the sound function than do higher level languages. We'll explore some of the really clever things you can do with sound in chapter 7. For now, we'll simply introduce you to the fundamentals of the sound mechanism.

To produce sound on the speaker you need to be familiar with some powerful assembly-language concepts. The first of these new concepts is communicating with the outside world using IN and OUT instructions. These instructions are the only way an assembly language program can communicate with those peripheral devices for which there are no DOS functions. In fact, the DOS functions themselves use IN and OUT to communicate with all the peripherals, including the screen and keyboard.

The other new concept we'll be exploring in this section is the use of the 8088 *logic instructions*. These instructions are common ones in assembly-language programs, and permit you to do logical manipulations on the bits in certain registers.

We'll show you our sound-producing program, and then explain how it makes use of these ideas.

The SOUND Program

Use the "A" command in DEBUG to type in the following program (or use "E" to type in the hex values shown below).

```
A>debug
-aløø
ø8F1:ø1øø in al,61
ø8F1:ø1ø2 and al,fc
ø8F1:ø1ø4 xor al,2
ø8F1:ø1ø6 out 61,al
ø8F1:ø1ø8 mov cx,14ø
ø8F1:ø1øB loop 1øb
ø8F1:ø1øD jmp 1ø4
ø8F1:ø1øF
```

Make sure it's correct with "U":

```
-uløø,1øe
ø9ø5:ø1øø E461          IN      AL,61
ø9ø5:ø1ø2 24FC          AND     AL,FC
ø9ø5:ø1ø4 34ø2          XOR     AL,ø2
```

```
0905:0106 E661          OUT     61,AL
0905:0108 B94001        MOV     CX,0140
0905:010B E2FE          LOOP    010B
0905:010D EBF5          JMP     0104
```

Before you run the program, save it to disk (you'll be sorry if you omit this step — don't say we didn't warn you):

```
-nsound.com
-rbx
BX 0000
:
-rcx
CX 0000
:f
-w
Writing 000F bytes
```

Now run it:

```
-g
```

Well, what do you know? A nice tone sounds. The only trouble is, you can't turn it off! Now that's a real inconvenience. Even the Alt Ctrl Del key combination — resetting the system — doesn't have any effect.

You have to actually turn the whole computer off and then on again to get rid of the sound and recapture control of the computer. At least, this is true if you execute the program from DEBUG. If you execute it directly from DOS, you *can* interrupt it with Alt Ctrl Del; but, when you execute it from DOS, the tone has an unpleasant burbling sound to it. In chapter 7 we'll learn why this is true, and in chapter 4 we'll learn how to guarantee that we can break into the program. But for now, let's content ourselves with trying to understand what our program does.

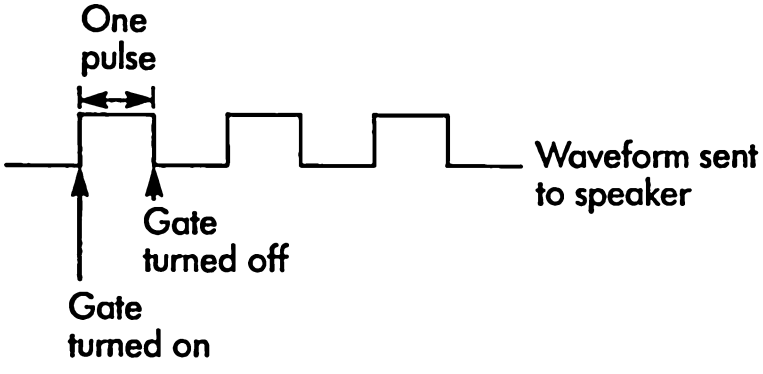
Fiddling with the Outside World

When we access most of the peripherals on the PC we can use DOS functions to help us out. You've already seen how this works with the Display Output function, which puts a character on the screen. When we use a DOS function we don't actually control the peripheral with instructions from our program: we let the DOS routine do that for us. This saves us a great deal of trouble and is generally just what we want.

When we access the speaker, on the other hand, we actually use instructions in our program to cause something to happen to a physical device. We can't use a DOS function, because there is no DOS function

for the speaker (the creators of PC-DOS must not have thought it was important). This gives us the opportunity to explore just how powerful assembly language can be, when it communicates directly with devices in the outside world.

To make sounds, our program actually turns on and off an electronic "gate" (which is a kind of switch). Each time we turn the gate on and then off again, we create a *pulse*: a brief period when current flows in a circuit. (See the illustration below.) These pulses are amplified and sent to the speaker, where they make a sound. This gate is turned on and off with the OUT instruction in the program above, as we'll explain soon.



The faster we send the pulses, the higher the pitch of the sound. We can control how fast we send the pulses by putting a *delay* into our program. We turn the gate on, delay, turn the gate off, delay, and so on. The LOOP instruction in the program above is used to cause the delay, as we'll see later.

The OUT Instruction

The instruction which turns the gate on and off is the OUT 61,AL in location 106 of our program. To understand what this instruction is doing you need to know about "Input/Output ports."

Ports are somewhat like registers, in that you send 8-bit or 16-bit data to them (from the AL or AX register). You can also read their contents back into the AL or AX register. However, the big difference between ports and registers is that *ports are connected to physical devices in the outside world.* (See Figure 3-5.)

So when you change something in a port, you are sending a message to some peripheral device, such as the video screen, disk drive, or in our case, the speaker. There can be, theoretically, up to 64K ports in the IBM PC. In reality only a small fraction of these are used, since there are usually less than a dozen peripherals connected to the PC.

OUT Instruction

Sends byte or word to input/output port.

To output a byte to port number PORTNO

```
OUT  PORTNO,AL
```

To output a word to port number PORTNO

```
OUT  PORTNO,AX
```

The port number can also be placed in the DX register, prior to executing the OUT

```
MOV  DX,PORTNO
OUT  DX,AL
```

Flags affected: none

The OUT instruction is something like MOV, except that it doesn't MOVe a byte into a register, it MOVes it (copies it actually) *from* a register *to* a port. The register the byte (or word) is to be moved from (it must be either AL or AX), and the number of the port to be moved to, are specified in the instruction. Thus,

```
OUT  61,AL
```

causes the contents of the AL register to be placed in port number 61. Figure 3-6 shows the operation of the OUT 61,AL instruction.

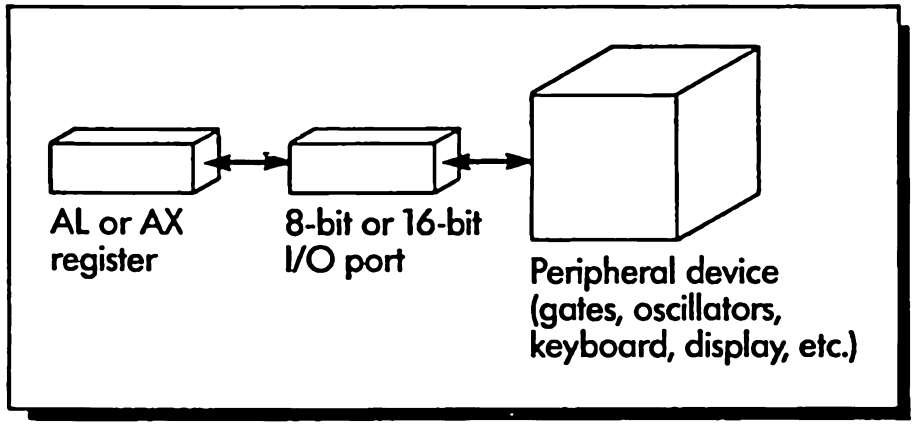
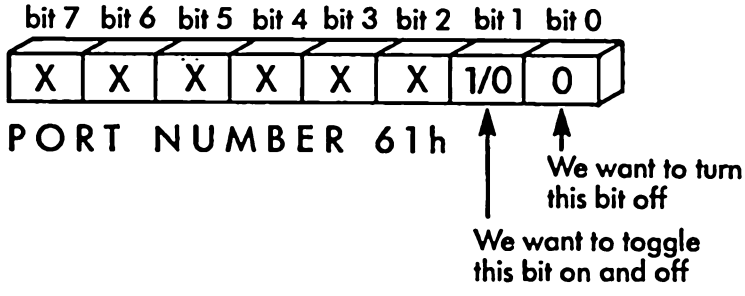


Figure 3-5. Input/output ports

Getting Down to Bits

In the case of beeping the speaker we're really only concerned with two of the bits in the byte we send to port number 61h. These are number 1 and number 0 (which as you recall from the bit numbering diagrams earlier in this chapter are the two bits on the right):



Bit 0 is connected to an oscillator which we want to turn off and leave off, so we want to set this bit to zero. (The oscillator is used in another method of sound generation which we'll learn about in chapter 7. For the time being we just want to deactivate it.) Bit 1 is connected to the gate which generates the pulses to be sent to the speaker, as we discussed. The other bits in this port (those marked "X" in the illustration above) do various other things, not related to the speaker (such as turning the cassette motor on and off), and for this reason *must not be changed.* Our goal is then to set bit 0 to 0, and turn bit 1 on and off just fast enough to make a nice tone in the speaker, but not change the other bits. How do we do that? We're going to need some more 8088 instructions.

The IN Instruction

In order to change bits 0 and 1 in port 61h without changing the others, we need to find out *how all the bits are initially set*: whether to 0 or to 1. Once we know how they're set, we can write the unchanged values of the ones we *don't* want to change back into the port, and at the same time write the new values of the ones we *do* want to change.

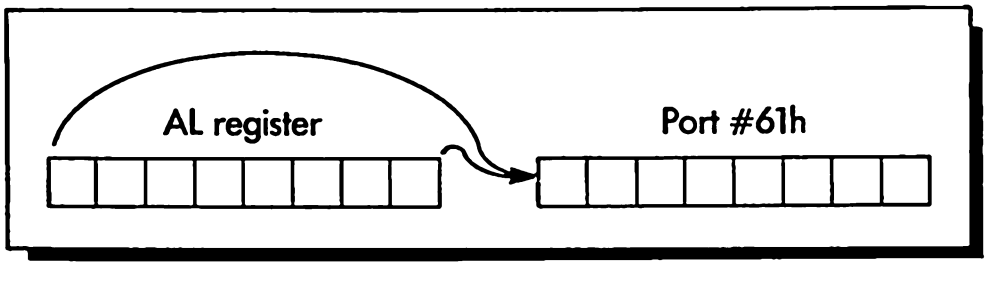


Figure 3-6. Operation of the OUT 61,AL instruction

IN Instruction

Receives byte or word from input/output port.

To input a byte from port number PORTNO

```
IN  AL,PORTNO
```

To input a word from port number PORTNO

```
IN  AX,PORTNO
```

The port number can also be placed in the DX register, prior to executing the IN

```
MOV  DX,PORTNO
IN   AL,DX
```

Flags affected: none

So how do we find out the initial values of the bits in the port? We use the IN instruction, which is the opposite of OUT. IN reads the byte *from* a port *into* a register. The register and the port number are specified in the instruction. Thus,

```
IN  AL,61
```

takes the byte in port 61 and reads it into the AL register. Figure 3-7 shows the operation of the IN AL,61 instruction.

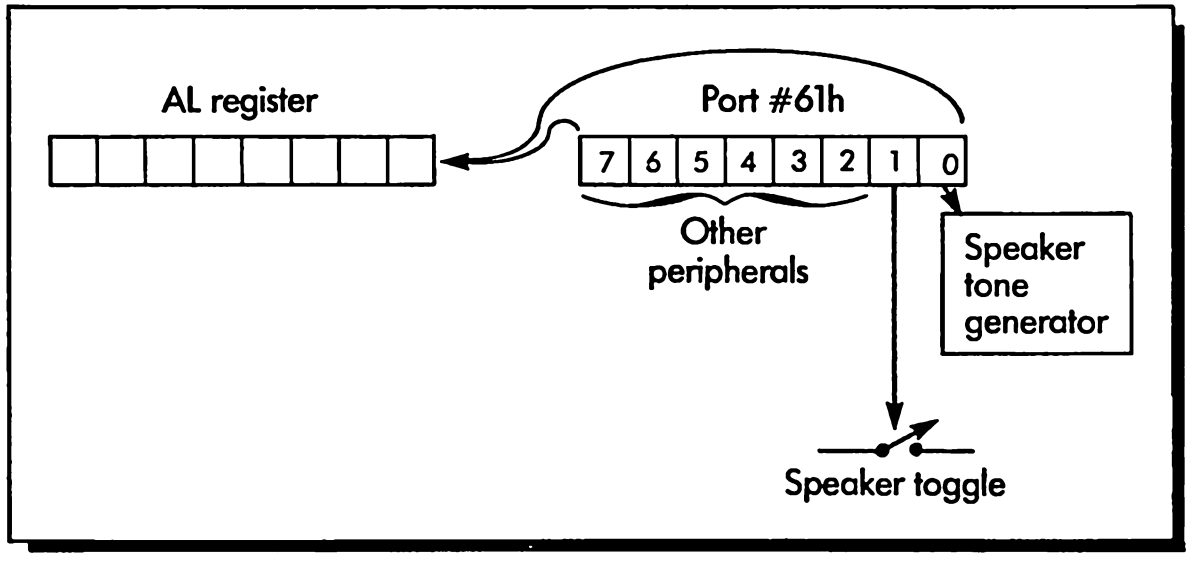


Figure 3-7. Operation of the IN AL,61 instruction

The AND Instruction

Once we've read the contents of port 61 into AL, we want to change bits 0 and 1 so we can send them back out to the port. This is a two-step process. First we get rid of the old value of these bits using an AND instruction, then we use a special instruction called XOR to "toggle" the bit connected to the *gate* which drives the speaker.

First we'll set the two unwanted bits to zero with an AND instruction.

AND Instruction

Performs logical "AND" on two operands. Result (conjunction) is stored in leftmost operand.

Register with register

```
AND AL,BL
AND BX,CX
```

Immediate with register or memory

```
AND DL,BYTE
AND MEM_WORD,WORD
```

Register with memory and vice versa

```
AND MEM_BYTE,DBYTE
AND DBYTE,MEM_BYTE
```

Flags affected: CF, OF, PF, SF, ZF

Flags undefined: AF

As you may recall from BASIC or some other higher-level language, ANDing two bytes together has the effect of turning off the bits in the result unless *both* of the corresponding bits in the two bytes are set to 1. If only one, or neither one, of the corresponding bits is set to 1, the resulting bit is set to 0.

We can summarize this in the following table:

0 AND 0 = 0

0 AND 1 = 0

1 AND 0 = 0

1 AND 1 = 1

As an example we'll AND two bytes together:

```
0 1 0 0 1 1 1 1   <— This
1 1 1 1 0 0 0 1   <— ANDed with this
-----------------
0 1 0 0 0 0 0 1   <— Gives this
```

If you're not familiar with the use of this operation you might want to work out a few more examples before going on. (Notice that nothing is *carried* to the adjacent column in logical operations: each column is a separate stand-alone calculation.)

AND can be used to "get rid of" (that is, set to zero) the bits we *don't* want in a byte, while at the same time keeping the bits we *do* want. This is often called "masking off" the unwanted bits. In our program we want to mask off the two least significant bits, 0 and 1, while keeping all the others. So we AND the byte in the AL register with the hex number FC, which is 11111100 in binary.

Suppose the number we read from the port is 4Dh, which is 01001101 binary. We mask off the two lower bits by ANDing on the FCh, which gives 70h, as shown in the figure below:

```
0 1 0 0 1 1 0 1   <— Number read from port
1 1 1 1 1 1 0 0   <— ANDed with this
-----------------
0 1 0 0 1 1 0 0   <— Gives this
```

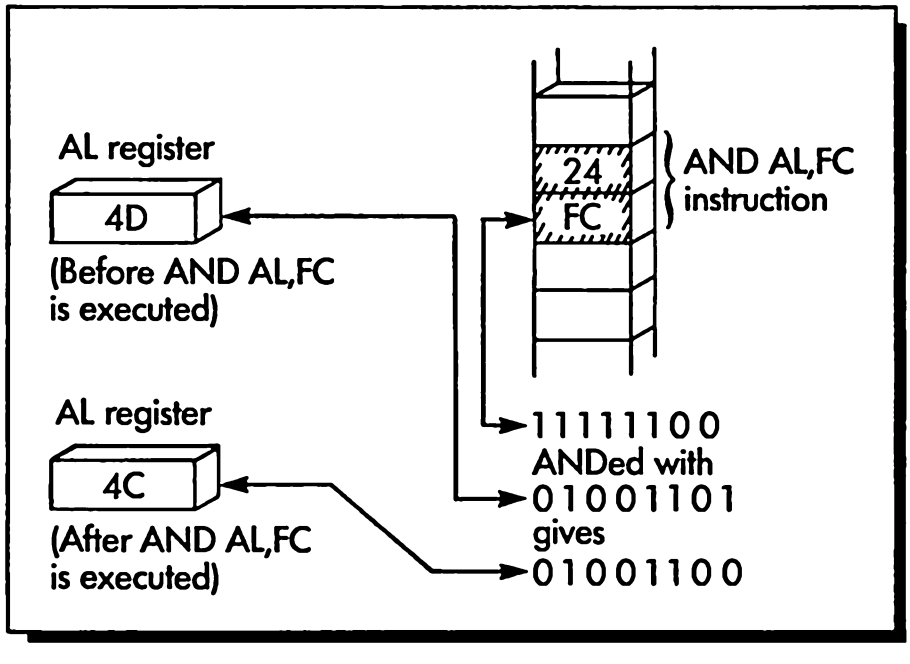


Figure 3-8. Operation of the AND AL,FC instruction

The operation of the AND AL,FC instruction is shown in Figure 3-8.

The XOR Instruction

Now that we've gotten rid of bits 1 and 0, we need a way to turn bit 1 on and off, over and over again, to generate our pulse train. What works nicely for this is a semi-magical instruction called XOR.

XOR Instruction

Performs logical Exclusive OR on two operands. Result (disjunction) is stored in leftmost operand.

Register with register

```
XOR AL,BL
XOR BX,CX
```

Immediate with register or memory

```
XOR DL.BYTE
XOR MEM_WORD,WORD
```

Register with memory and vice versa

```
XOR MEM_BYTE,DBYTE
XOR DBYTE,MEM_BYTE
```

Flags affected: CF, OF, PF, SF, ZF

Flags undefined: AF

We call the XOR instruction "semi-magical" because if something is on, XOR can turn it off, and if something is off, XOR can turn it on. How? XOR stands for "Exclusive OR," which means "either one or the other, but not both." In terms of how it operates on bits, XOR looks like this:

0 XOR 0 = 0

0 XOR 1 = 1

1 XOR 0 = 1

1 XOR 1 = 0

For example,

```
1 1 0 0 0 0 1 1   <— This
0 0 0 0 1 1 1 1   <— XORed with this
-----------------
1 1 0 0 1 1 0 0   <— Gives this
```

Notice how a 1 XORed with a 1 is 0, while a 0 XORed with a 1 is a 1. If we repeatedly XOR a 1 with another bit, that bit will turn on, then off, then on, and so forth; as shown in the diagram below:

```
0 XOR 1 = 1
1 XOR 1 = 0
0 XOR 1 = 1
1 XOR 1 = 0 <— This toggles back and forth
↑
This is the switch
This toggles back and forth
```

No such toggling action takes place when we XOR a 0 instead of a 1. In fact, XORing a 0 to another bit leaves the other bit unchanged:

```
This is the same   as this
↓                  ↓
0 XOR 0 = 0
1 XOR 0 = 1
This leaves bit unchanged
```

In our particular case what we want to do is turn on and off bit 1 in the AL register, so we XOR AL with the hex number 2, which is 00000010 binary. This leaves all the bits except bit 1 unchanged, while if bit 1 was a 0 it now becomes 1, and if it was a 1 it becomes 0. The operation of the XOR AL,2 instruction is shown in Figure 3-9.

So in our program all we need to do is put this XOR instruction in a loop along with the OUT instruction, and then every time we go through the loop we'll change bit one in port 61 from 1 to 0 or from 0 to 1, thus "toggling" (switching) it repeatedly on and off.

The Time Delay

The only thing left to explain about this program is the time delay in lines 108 and 10B. A delay is necessary in the loop that toggles bit 1 on and off, because the computer executes its instructions so very rapidly compared with the frequency of sound. The 8088 can whiz through our little program so fast, in fact, that the tone generated in the speaker would be far too high to be heard by human ears (or even dog ears).

So we need to slow things down. We do this by setting up a LOOP instruction to cause a delay. This is done by putting the appropriate-sized count in the CX register, and then simply executing the LOOP instruction that many times. The LOOP is written to jump to itself until the count in CX goes to zero.

```
Ø1Ø8  MOV  CX,14Ø      — Sets count of 14Øh into loop counter
Ø1ØB  LOOP Ø1ØB        — Jumps to itself 14Øh times
```

By trial and error, we can determine that the length of time taken by the LOOP instruction, times 140h (320d), produces a delay just long enough to make a tone with a pitch in the audible range.

When the LOOP instruction has finished jumping to itself, JMP 104 is executed and control goes back up to toggle the bit again. The effect is bit-on, delay, bit-off, delay, and so on.

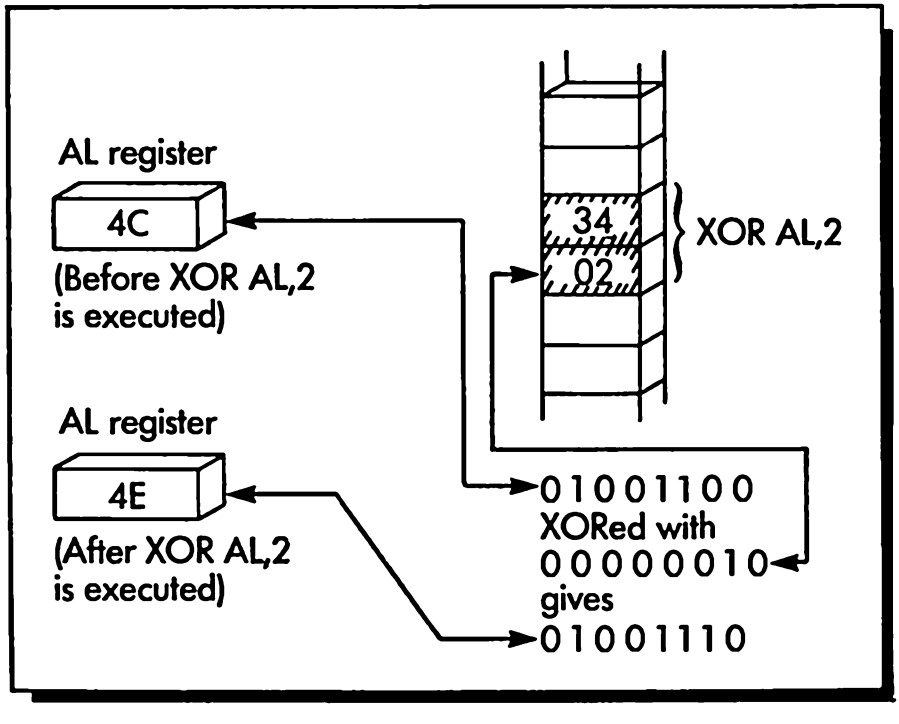


Figure 3-9. Operation of the XOR AL,2 instruction

Here's an annotated version of the program to summarize how the various instructions work together:

```
-u1ØØ,1Øe
Ø9Ø5:Ø1ØØ E461          IN      AL,61       <— Get old value from I/O port
Ø9Ø5:Ø1Ø2 24FC          AND     AL,FC       <— Mask off lower two bits
Ø9Ø5:Ø1Ø4 34Ø2          XOR     AL,Ø2       <— Toggle bit 1 (on or off)
Ø9Ø5:Ø1Ø6 E661          OUT     61,AL       <— Send result to port
Ø9Ø5:Ø1Ø8 B94ØØ1        MOV     CX,Ø14Ø     <— Set up delay of 14Øh cycles
Ø9Ø5:Ø1ØB E2FE          LOOP    Ø1ØB        <— Repeat this instruction 14Øh times
Ø9Ø5:Ø1ØD EBF5          JMP     Ø1Ø4        <— Go back to toggle again
```

That's all there is to the program. It's a lot like someone standing next to a wall switch, flicking it on and off as fast as they can — except, of course, that the program is faster than the fastest human fingers, and the switch is connected to a speaker instead of a light.

Changing the Pitch

Want to change the pitch of the sound generated by our program? All you have to do is change the number you load into the CX register. This changes the delay, which changes how rapidly the gate is toggled, which changes the frequency of the sound. Smaller numbers will cause less delay, which will increase the frequency and generate a higher tone. Larger numbers will lower the tone.

Let's raise the tone a bit by changing the 140h to 100h.

```
Ø9Ø5:Ø1Ø8 B94ØØ1    MOV  CX,Ø14Ø
             ↑
          Change this from 4Ø to ØØ to increase pitch
```

Load the program from DEBUG (unless the program is still in memory, of course) and use "E" to change the 40 in location 109 to 0 (this will change the 140h to 100h).

```
A>debug sound.com
-e1Ø9
Ø9Ø5:Ø1Ø9  4Ø.Ø
```

The resulting program is identical to the first one, except for the one changed byte:

```
-u1ØØ,1Øe
Ø9Ø5:Ø1ØØ E461          IN      AL,61
Ø9Ø5:Ø1Ø2 24FC          AND     AL,FC
Ø9Ø5:Ø1Ø4 34Ø2          XOR     AL,Ø2
Ø9Ø5:Ø1Ø6 E661          OUT     61,AL
Ø9Ø5:Ø1Ø8 B9ØØØ1        MOV     CX,Ø1ØØ     <— Shortened delay raises pitch
Ø9Ø5:Ø1ØB E2FE          LOOP    Ø1ØB
Ø9Ø5:Ø1ØD EBF5          JMP     Ø1Ø4
```

Now run the program again. You should hear the difference in pitch.

```
-g
```

Of course, you have to restart your whole system after this experiment, so it's not a convenient program to experiment on very much. Later we'll show you how to transform it into a more useful program.

Summary

In this chapter we've talked some more about how assembly language uses the computer's memory and registers. You've learned how to examine and modify the 8088's main registers — AX, BX, CX, and DX — using DEBUG's "R" command; how to save a program on disk using the "N" and "W" commands, and how to get it back again using "N" and "L." You know how to make the speaker produce a tone. And finally, you've learned some more 8088 instructions: INC, LOOP, IN, OUT, AND, and XOR.

4

Inside DOS—The Disk Operating System

Concepts

The purpose of DOS
The different parts of DOS
The IP register
Memory buffers
Indirect addressing
Using the BX register as a pointer
Sending messages to the printer
Sending control codes to the printer

Debug Commands

RIP To change to IP register

8088 Instructions

DB = Define byte to assemble strings (pseudo-op)

DOS Function Calls

Keyboard Input
Print String
Buffered Keyboard Input
Printer Output

Applications

EMPHAP program — Turns on printer's "emphasized" print
NORMALP program — Restores printer's normal print

In the IBM PC, as in most modern computers, there is an intimate connection between the assembly-language programs which run in the computer, and DOS — the Disk Operating System. In this chapter we're

going to talk about DOS, what it does, and how it relates to assembly language. We'll also write some programs that will extend your understanding of this relationship and teach you more about assembly language.

What Is a Disk Operating System?

You're probably already aware of many of the user-level functions of the DOS on your PC. Whenever you see the "A>" prompt it is DOS that has printed it, and when you type in a command like DIR or COPY, it's DOS that carries out the command. Also, when you type a program name like ASM or BASICA, it's DOS that finds the program and loads it into memory, and is waiting there to resume control when your program is finished.

So one of the primary purposes of DOS is to manage other programs, by keeping them on the disk in such a way that they can be called by name, loaded into memory, and executed; and by providing functions to permit you to list, copy, and erase these programs or data files.

These "file management" operations are an essential part of DOS, but they are not the whole story. Beneath the file management part of DOS is another, more sophisticated level, which can only be reached through assembly language. What is this deeper level of DOS, and what does it do?

The Historical View

The earliest operating systems performed only the file management functions, and provided no further interaction or assistance to other programs using the system, once they were loaded. Thus, if you wanted to write an assembly language program to, say, put a character on the video screen, you had to figure out exactly how the video circuitry worked, and then go through a complex series of instructions to tell this circuitry where to put the character. Similarly, if you wanted to write a file to the disk, you had to understand the most minute details of the disk operation, such as where every byte was located on the disk, how fast the disk was spinning, and how long it took the stepping motor to reach different tracks. As you can imagine, this made programs very long and complex. Figure 4-1 shows schematically what this looked like.

These early operating systems had another disadvantage, too. If you physically interchanged your video terminal, your disk drive, or some other device, for one of a different kind, then you had to rewrite all the programs that used these devices, since the instructions in your program

that worked for one kind of device would not work for another that was even slightly different. Worst of all, your programs would only run on other computers which were exactly the same as yours: same video terminal, same disk drives, same everything. It was impossible to transport a program from one brand of computer to another. All this was very inconvenient.

DOS to the Rescue

Then someone had a very clever idea. This idea depended on the fact that the routines to access the peripheral devices — the video terminal, the disk drives, and so on — *were already in the disk operating system.* They had to be there, because DOS needed to interact with these peripherals. The clever idea was this: Why not make these routines accessible to other programs? That way, if you wanted to, say, write a character to the video screen, you wouldn't have to know anything about the video circuitry. All you would need to know was the entry point of the video routine and how

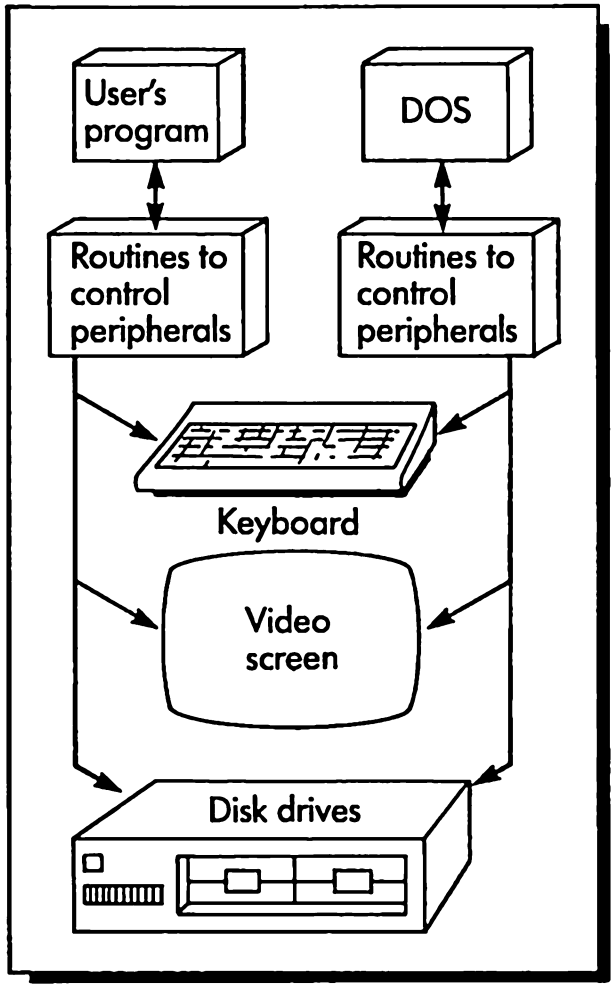


Figure 4-1. Old-fashioned operating systems

to tell it what character to print. Then you could let the DOS routine worry about all the tedious hardware-dependent details. Figure 4-2 shows a modern operating system, which lets the user program make use of its input/output routines.

Does this remind you of anything? Have you realized that you've already written programs that use routines in DOS? The happy face programs use the Display Output function call to print characters on the screen. This function call required only three instructions:

```
MOV DL, 1       ←— Put ASCII character in DL register
MOV AH, 2       ←— Put DOS function number in AH register
INT 21          ←— Interrupt #21 call to DOS
```

Putting a character on the screen would have required dozens of assembly-language instructions if we had written the routine to do it ourselves, as we would have had to do with the old-fashioned kind of operating system. We would have needed to worry about such topics as

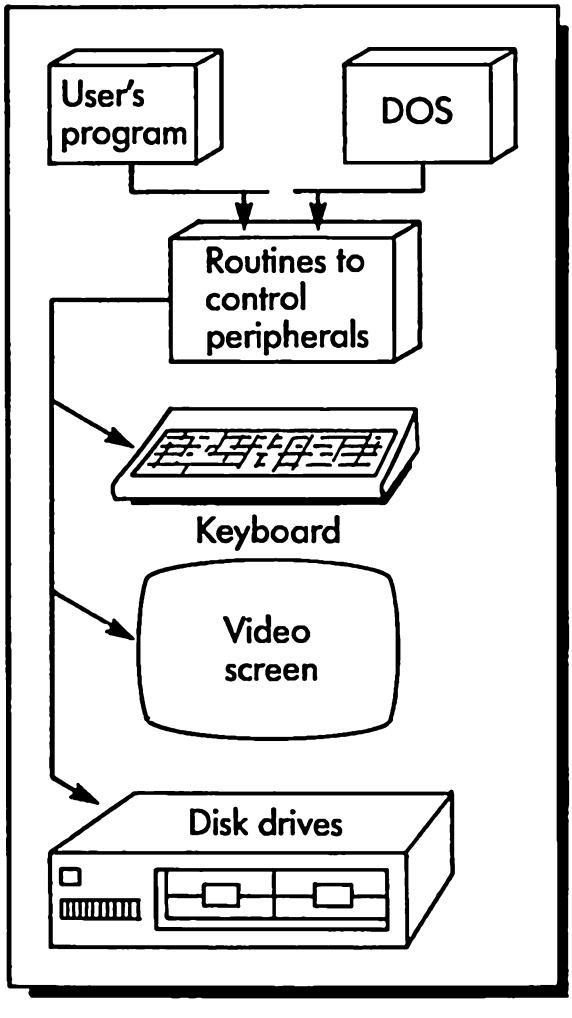


Figure 4-2. Modern operating systems

what mode the display was in, what the horizontal retrace was doing, whether the character was a linefeed (if so, we'd need to move the cursor down a line), whether we were on the bottom line (if so, we might need to scroll the screen up), and so on. But by simply calling a routine in DOS, we have changed our task from an extremely complex one, requiring detailed understanding of the computer's hardware, to a comparatively simple one needing only a few facts about the operating system, and only three instructions.

Using the Speaker — No Help from DOS

Remember the routine we wrote in the last chapter to make a "beep" sound on the speaker? This program provides a small example of the difficulties involved in writing our own routines to access a peripheral. In this routine we had to figure out all sorts of details, such as how long a delay loop to make to produce a given pitch. The resulting program was seven instructions long. If there were a DOS function call to perform this function (which there isn't), it would require no detailed understanding of how the speaker works, and could get by with only two instructions:

```
MOV  AH, 99        ←— Hypothetical number of BEEP function
INT  21            ←— Call DOS
```

Of course, beeping the speaker is one of the simplest I/O jobs we can perform. The advantages to be gained by using DOS routines are much greater for other peripherals, such as the keyboard and disk drive, as we'll see.

Program Transportability

Besides the convenience of being able to write shorter programs and not needing detailed knowledge of how to program peripherals, there is another big advantage to letting DOS do our input/output. Our program will work even if we replace some of these peripherals — like the video terminal or the disk drives — with completely different models from different manufacturers.

In fact, our program will even work on an entirely different computer, provided it uses the MS-DOS operating system. Since MS-DOS is very similar to PC-DOS (as we noted in the Introduction), you can take your happy face program and run it on any of the so-called "IBM compatible" computers that use MS-DOS. The DOS function calls will have the same numbers, and be accessed in the same way, so your program will operate just as before. On a small program like "happy face" this is hardly an

earth-shaking issue, but if you have invested thousands of hours in a sophisticated accounting or word-processing program, it's nice to know that it can be used on a variety of different computers, with little additional programming investment. It's also nice to know that what you learn in this book is applicable to other computers besides the IBM.

Something Has to Change

Of course, *something* has to change when you try to run the same program on a computer with different peripherals, or on a different make of computer. What changes are the input/output routines, buried somewhere in DOS, which actually communicate directly with the physical device. Thus, if you got a different kind of disk drive, or video terminal, or wanted to use the operating system on a different computer altogether, then you would need to change your operating system to work with this new device. Actually, only part of DOS needs to be changed when these routines are changed, the part called IBMBIOS.

We're going to learn more about IBMBIOS and the other parts of DOS in a moment. First, however, let's explore another example of a DOS function call, so you can begin to see the variety of different things these calls can do for your programs.

The KEYBOARD INPUT Function

KEYBOARD INPUT Function — Number 01h

Enter with:

Reg AH = 1

Execute:

```
INT 21
```

Return with: keyboard character in Reg AL

Comments: Ctrl Break causes exit from function

You might think it would be a comparatively simple task to read a character from the keyboard into your program. Actually, it is *if* you use the DOS function call we're about to describe. If you wanted to write the code to do it yourself, it would take ten pages of code! How do we know?

Because that's how much code IBM used in the ROM routine built into the PC, as you can see by looking at appendix A in the *IBM Personal Computer Technical Reference* manual.

What does all this ROM code do? Well, for example, it has to figure out if the Alt or Shift or Ctrl keys are pressed, and what this means when combined with other keys. It has to know what to do if Ctrl Break or Alt Ctrl Del are pressed. It has to store normal key entries in a buffer (an area of memory), so that if your program is busy doing something else while you are typing, no keystrokes will be lost. If this buffer gets full, the routine has to sound the beeper to let you know. And so on, and so on. Aren't you glad you don't have to figure all this out every time you want to read a character from the keyboard?

We'll be talking more about these ROM routines in chapter 9. Until then, all you need to know about them is that there are routines built into ROM to help with input/output, and that DOS makes use of these routines to simplify assembly-language programming.

Here's a short program that makes use of the Keyboard Input function. Get into DEBUG, and type the following:

```
A>debug
-a100
08F1:0100 mov ah,1
08F1:0102 int 21
08F1:0104 int 20
08F1:0106
```

← Enter these instructions

← Press ↵ to leave "A" command

This is an even shorter program than the happy face one! To make sure it's accurate, unassemble it with "U":

```
-u100,105
08F1:0100 B401          MOV     AH,01
08F1:0102 CD21          INT     21
08F1:0104 CD20          INT     20
```

Now, to execute this program, you type "G". Uh, oh — nothing seems to be happening. The computer is just sitting there. Is it stuck? No problem. Just press any key, "z" for example.

```
-g
z
Program terminated normally
-
```

The computer comes back to life, and you're in DEBUG again. What

was all that about? Nothing mysterious. When you started the program, the first instruction put a 1 in the AH register to tell DOS that we wanted to execute the Keyboard Input function. Then INT 21 called DOS (as you know), which took us straight to the routine to read a character from the keyboard. The function waits until something is typed before it lets the program go on, so until we hit a key the program sits there, looping endlessly in the DOS routine.

Once we strike a key, the function terminates, and the next instruction in our program is executed, which is the INT 20, which terminates the program and returns us to DEBUG. The ASCII value of the character is also returned in the AL register, although this short program does not make use of that fact.

Potential Trouble

There's one character which will cause things to act a little differently if you type it in this program: the Ctrl Break key. Let's see what happens:

```
-g                  ← Run the program
z                   ← Type a normal character
Program terminated normally
-g                  ← Run the program again
^C                  ← Type Ctrl Break
```

```
AX=Ø1ØØ  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1Ø4   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1Ø4 CD2Ø          INT     2Ø
```

Wow! You get the printout of all the registers that you got before by typing DEBUG's "R" command. And now, try typing "g" to run the program again:

```
-g                  ← Run it
                    ← It doesn't wait for you to type something!
Program terminated normally
-g                  ← Try it again
                    ← Same result
Program terminated normally
```

Something's gone wrong with the program. It no longer waits for our input from the keyboard when we type "G" to run it; it says "Program terminated normally" immediately. Why is that?

The Instruction Pointer Register

Look closely at the register display we just saw. There's a new part of this display you should learn about, in order to understand where our program went awry. In the middle of the middle row it says "IP=0104." Why is this important? To understand its significance, you need to know that the 8088 keeps track of where it is in a program by keeping the *address of the instruction currently being executed in the IP register.* The IP register is a 16-bit register something like AX, BX, and so on, except that it is used *only* to hold the address of the current instruction. Each time an instruction is executed, the 8088 updates the IP register to point to the next instruction.

> The 8088 microprocessor keeps track of where it is with the Instruction Pointer (IP) register.

Thus, at the beginning of our program, IP contains 100, since that's where all programs are supposed to start in DEBUG. In fact, DEBUG puts this value into IP when it's first loaded, as you can see by loading DEBUG and typing "R" immediately. After we execute the first instruction in our program, the IP contains 102, since that's the address of the next instruction. And finally, for the last instruction, it contains 106. When the program terminates with an INT 20 instruction, DEBUG automatically sets the IP register back to 100, so that it's ready to start the program again.

Now, the reason our program doesn't work the way it should is this: when you hit Ctrl Break, DEBUG terminated the program right in the middle, just before the program had a chance to execute the INT 20 instruction. The instruction shown in IP in the register display is the one *about to be executed.* So, when we return to DEBUG from our program, the IP contains 104, not the 100 that it should. Since the program has not terminated with an INT 20 instruction, the IP will not be reset to 100. So when you type "G", DEBUG will start the program at whatever address is in the IP register. If 104 is in IP, then that's where the program will start. But the only thing at 104 is an INT 20, which will terminate the program and bring us straight back to DEBUG with "Program terminated normally." The call to the Keyboard Input function will never be executed.

How can you start over at the beginning of the program? It turns out you can modify the contents of the IP register with DEBUG, just as you

can the AX and other general purpose registers. Enter "R", followed by "IP".

```
-rip                ←— You type this, to see the IP register
IP Ø1Ø4             ←— Contents of IP
:1ØØ                ←— Type this to change it to 1ØØ

-g                  ←— Now try the program again
z                   ←— It waits for you to type a character!
Program terminated normally
```

So we've fixed it! The moral is that it's only when your program starts at 100 and terminates itself with an INT 20 that DEBUG will automatically reset the IP to 100. If you start the program somewhere else, or terminate it in the middle, then you can't be sure what may be left in the IP. To avoid problems, get in the habit of checking the IP by typing "RIP", and setting it back to 100 if necessary, before you type "G" to run a program.

Typing in a Sentence

Suppose we wanted to use the Keyboard Input function to type in something longer than a single character. As you might guess, we can simply change the INT 20 to a jump back to the beginning of the program: JMP 100. Here's what you type in:

```
-aØØ
Ø8F1:ØØØ mov ah,1
Ø8F1:ØØ2 int 21
Ø8F1:ØØ4 jmp ØØ
Ø8F1:ØØ6
```

And here it is disassembled with "U":

```
-uØØ,Ø5
Ø8F1:ØØØ B4Ø1          MOV     AH,Ø1
Ø8F1:ØØ2 CD21          INT     21
Ø8F1:ØØ4 EBFA          JMP     ØØØ
```

Now when we run the program we can type in a whole sentence. While you're typing you can experiment with some of the editing features built into this system call. For instance, if you make a mistake, you can back-space. If you type Ctrl J (the "J" key pressed while the Ctrl key is held down) you'll get a linefeed. And if you hit ↵, the cursor will return to the start of the line, although you will still be in the

function. But if ↵ does this, how do we escape from our program? Try Ctrl Break — without further ado you'll be back in DEBUG:

```
-g
Now is the time for all good men to come to the aid of their country.
^C

AX=Ø1ØD  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1Ø4   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1Ø4 EBFA          JMP     Ø1ØØ
```

Actually it's not quite as clean as this, because the ^C prints over the first part of the phrase you typed in:

```
-g
^Cw is the time for all good men to come to the aid of their country.
```

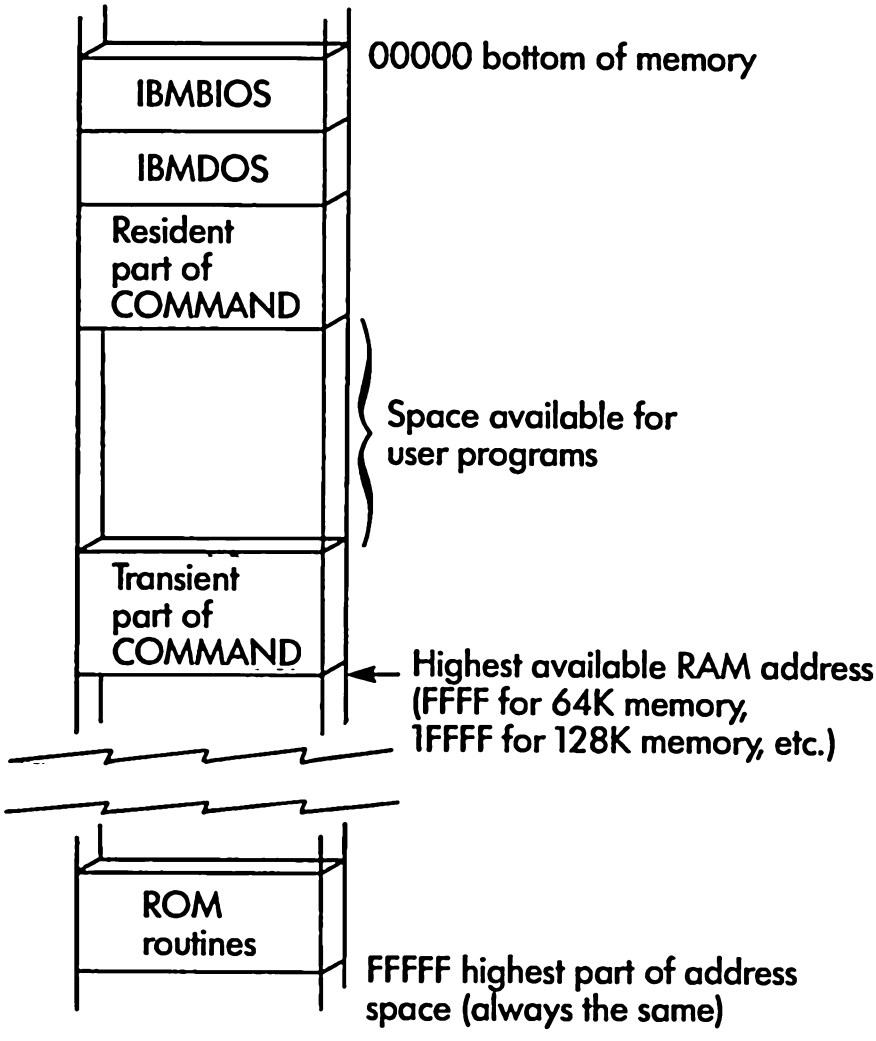
Notice how the registers have all been printed out again, as they were when we typed Ctrl Break in the single-key program above. Again, the IP contains 104. But this time it doesn't matter if we set it back to 100 or not: since the program consists of an endless loop, we can get into it anywhere without changing its operation.

You may be concerned that the programs we've used to demonstrate these functions so far don't seem to do anything very useful. Don't worry. At the end of this chapter, and in the next chapter, we'll combine the function calls we've learned into larger programs that will actually perform useful services and amaze your friends. Now, however, let's go back and talk about the various parts of DOS, and where these function calls fit into the overall DOS organization.

The Parts of DOS

Earlier we mentioned ROM and IBMBIOS, and said that they were parts of the Disk Operating System. Let's stop a minute now and describe the major parts of DOS. This will give you a rough idea of what the various parts of DOS do, and where they fit in the computer's memory. Be aware, however, that you really don't need to know a great deal about the internal workings of DOS to write programs in assembly language. So don't worry if some of the details of its operation seem a little vague at this point; you'll learn more about the operating system as we go along.

DOS is divided into four major parts: ROM, IBMBIOS, IBMDOS, and COMMAND. They are loaded into memory like this:



Notice that the lowest addresses are shown at the top of the diagram. This may seem backwards, but it's the way program listings are written, and it's the way IBM does it, so for consistency we're going to follow this format too.

To understand the roles played by the various parts of DOS, it's helpful to think of the entire operating system as some sort of large industrial corporation — we could call it "DOS Incorporated." The different parts of the system then correspond (very roughly) to the different management levels in the corporation.

The Workers: ROM (Read Only Memory)

ROM stands for "Read Only Memory." It corresponds to the blue-collar workers down on the floor of the factory, getting the actual work done. In DOS this work might be sending characters to the display screen, reading information from the disk drive and the keyboard, and so forth. By getting the work done we mean that the routines in ROM send instructions to peripheral devices such as the keyboard and disk drive

that actually do things in the outside world. This is the point where software "interfaces" (connects with) hardware.

The "products" that the ROM routines are producing are generally concerned with moving information from hardware to software and vice versa: reading a character from the screen into memory, sending a group of data from memory to the disk, and so on. In other words, ROM contains most of the actual input/output routines that communicate with the peripheral devices connected to the PC.

ROM is an actual physical part of the computer, a kind of memory like the RAM (Random Access Memory) you store your program in, except that the programs in ROM are installed by IBM at the factory and can't be changed. (They also don't vanish when you turn off the computer, the way programs stored in RAM do.) Since ROM is part of the physical computer it is documented in the *IBM Personal Computer Technical Reference* manual, which describes the physical characteristics of the machine, rather than in the *IBM Personal Computer Disk Operating System* manual.

You might not think of ROM as being part of DOS, since it exists even in cassette-based IBM PCs that don't have any disk drives. However, ROM contains routines to access the disks as well as the other peripherals, and when DOS is loaded from the diskette, the routines in ROM become an integral part of the operating system.

The remaining parts of DOS come on the DOS diskette, and are loaded in from the diskette when you initialize your system, either by turning it on, if it's off, or by hitting Alt Ctrl Del.

The Foreman: IBMBIOS

IBMBIOS supervises the activities of the ROM routines. If IBMDOS or another program wants to use a routine in ROM, the request is "passed through" IBMBIOS. That is, the request goes to IBMBIOS, which decides what to do with it before passing it on to the appropriate ROM routine. This has several advantages. If IBM discovers a mistake in the ROM, or if they want to modify it for some reason, they can't actually change the ROM (at least in those computers that have already been sold), since the ROM is a permanent part of the computer. But they can change the DOS diskette, which contains IBMBIOS, so that it incorporates the changes. This is like a human foreman who has learned so well what mistakes his employees are likely to make that he can compensate for them in the finished product.

Thus by issuing a new operating system with the revised IBMBIOS, IBM can in effect change the input/output routines in ROM, even though

ROM itself is unchanged. (It's modification of this sort that led to new revisions of the operating system being issued, such as when DOS 1.00 became 1.10, and so on.) Also, various error situations which can occur when an I/O routine is in use can be dealt with more flexibly if they are not a permanent part of ROM.

Management: IBMDOS

IBMDOS concerns itself with more general, less detailed problems than do ROM and IBMBIOS. You can think of it as the management part of DOS, having a larger perspective than the workers or the foreman. For instance, ROM and IBMBIOS know how to write a particular *sector* (a small amount of disk information) to the disk, but IBMDOS knows what entire *file* is to be written to the disk, and keeps track of what sectors have been written so far and where they are on the disk. (Don't worry, we'll be talking more about sectors and files, among other things, in the chapters on the disk system.)

IBMDOS also contains the "entry points" for the DOS function calls discussed previously, like the Display Output and Keyboard Input functions we've already used. (Entry points are simply addresses where these routines begin in memory.) It's this part of DOS that our assembly-language programs will be communicating with when they need to perform any input or output operations. The actual input/output routines may be in ROM, but your program must go through IBMDOS to use them, just as in a corporation we wouldn't place an order for 1000 widgets with the workers in the assembly line; we'd talk with some management-level people on a higher floor.

Chief Executive Officer: COMMAND.COM

COMMAND.COM is responsible for controlling the overall activities of the operating system. It's the part of DOS that prints the A> prompt and then figures out what to do with what you type in. You might say it is the intelligent part of the operating system. The other parts merely do what they're told, either by COMMAND.COM, or by another assembly-language program.

COMMAND.COM actually comes in two parts: a resident part, which lies just above IBMDOS in low memory, and a transient part that sits all the way at the top of memory, up to FFFF if you have 64K, up to 1FFFF if you have 128K, and so on. (Notice the difference between the memory you actually have, which might be say, 128K, and the entire addressable memory space in the computer, which is one megabyte, or 1,000K, with a high address of FFFFF.)

"Resident" means that this part of COMMAND.COM remains in memory at all times. The resident portion of COMMAND.COM contains basic control functions and error-handling routines. The transient portion communicates with users via the A> prompt, and contains the internal DOS commands like DIR, TYPE, and COPY. The transient part of COMMAND.COM can actually be written over by user programs if they need a lot of memory space. It is then loaded back into memory from the diskette by the resident portion when the user program is finished.

Acting Chief Executive Officer

When we write an applications program in assembly language (or in a higher-level language like Pascal, which is then compiled into machine language), and then execute it, this program takes over temporarily from the COMMAND.COM program, and assumes command of the computer itself. It then has access to all the facilities provided by IBMDOS, IBMBIOS, and ROM, just as COMMAND does when it's in charge. It can use these resources for its own purposes, and COMMAND can only regain control when the program is over, as when it executes the INT 20 interrupt.

Chairman of the Board

And who, you might ask, tells COMMAND.COM what to do? Why, you do — whenever you type a command following the A> prompt. Was it not this opportunity to exercise corporate power that convinced you to buy a computer in the first place?

Figure 4-3 gives some idea how the various parts of DOS fit together.

DOS Functions

We learned above that the DOS functions are input/output routines located in the ROM and IBMBIOS portions of DOS. They are accessed by making interrupt calls in the form of INT 21 to the IBMDOS part of the operating system, which then passes our request on to the appropriate routine in IBMBIOS or ROM. The particular function to be used is selected, as we've seen, by placing a particular number in the AH register before making the INT 21 call to DOS.

In chapter 2 we used the Display Output DOS function to write a happy face and other characters on the screen, and in this chapter we used the Keyboard Input DOS function to get a character from the

keyboard. What other DOS functions are there?

The most complete description of these functions is given in appendix D of the *IBM Personal Computer Disk Operating System* manual, which comes with your copy of DOS. In DOS version 1.10 the functions start with 0 and go up to number 2Eh. We wind up with a total of 41 functions (not all the available numbers were assigned). DOS version 2.00 uses 74 functions, and there is no reason why new versions of DOS will not contain even more. It might be educational for you to look through this appendix, just to get a rough idea of the kinds of things these calls do. Many of the descriptions will be mysterious to you at this point, but by the time you finish this book you will be reading appendix D for relaxation, like the Sunday comics.

Since there are so many functions we are not going to provide detailed descriptions of them all in this book. Instead, we will concentrate on the most commonly used ones, and those that most easily demonstrate how particular parts of the operating system work. Once you know these, you should be able to figure out how the others work, since there are many similarities.

DOS functions can be divided roughly into two categories: those that

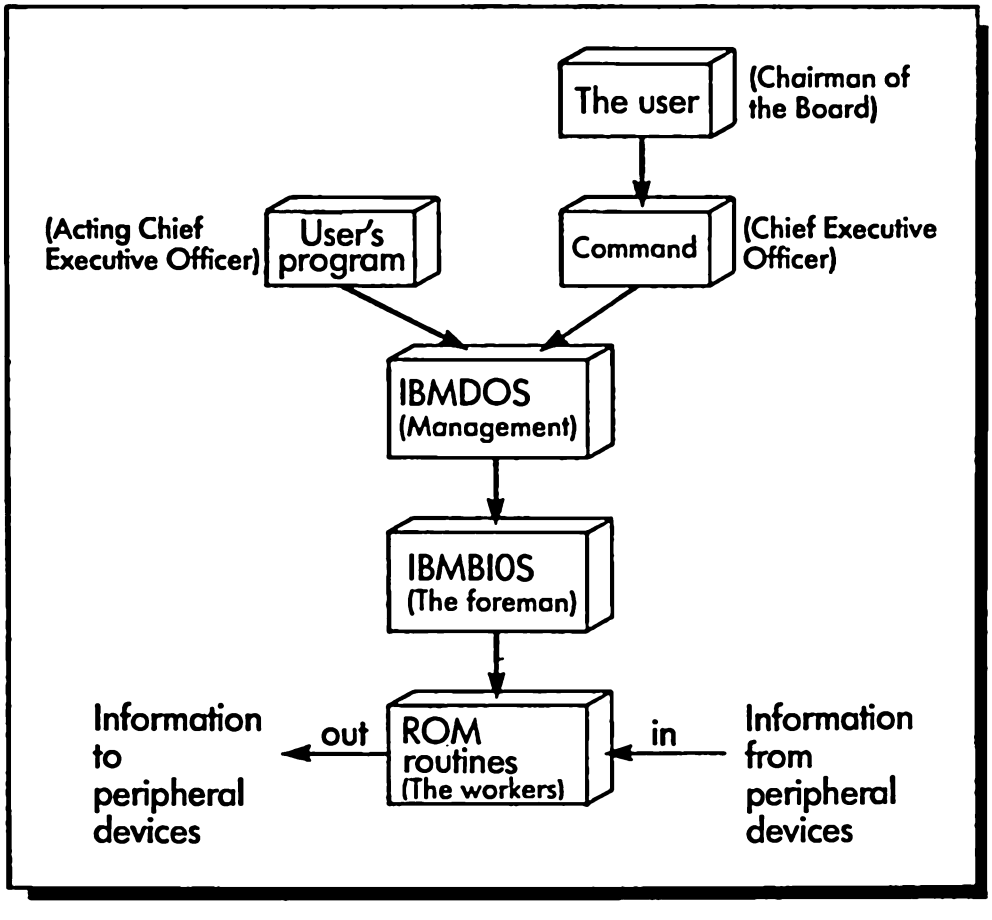


Figure 4-3. Organization of DOS

deal with the disk, and those that deal with other peripherals, such as the video screen, keyboard, and printer. The non-disk functions are generally simpler, so we will cover several more of them in this chapter. We'll discuss the disk functions in chapters 11 and 12.

The Print String Function

Let's start off by learning a new DOS function: one that prints a string of characters.

PRINT STRING function — Number 09h

Enter with:

Reg AH = 9

DS:DX = address of start of string

Execute:

```
INT 21
```

Comments: string must terminate with "$" (dollar sign)

You may have noticed something new in the box above: the expression

```
Reg DS:DX = address of start of string
```

This means that the function needs both the segment address and the offset address of the string, and that the segment address is to be placed in the DS register and the offset address is to be placed in the DX register. You don't need to know about the DS register yet. DEBUG (or DOS) takes care of making sure the correct value is in this register, so for the time being you can ignore it. Later, in chapter 8 on memory segmentation, we'll find out about the "Segment Registers," of which DS is one.

We already know how to print a single character on the screen, using the Display Output function. That's good as far as it goes, but many times in a program we'd like to display a whole string of characters at once. Print String lets us do just that.

Here's how it works. Before you can use this function you need to put the string — consisting of the actual characters you're going to print —

somewhere in memory. (Makes sense, doesn't it? Can't print them if they're not there.) The string consists of ASCII characters, and it *must end with a dollar sign ($)*. The dollar sign is the only way the function knows when it has come to the end of the string, so it's important that you don't forget it.

> Strings to be printed by the Print String function must end with a dollar sign.

To use the Print String function you first put the starting address of the string in the DX register. Next, you put the function number 9 in the AH register, and finally you call DOS with an INT 21. Let's write a program that makes use of this function to print a string.

```
A>debug
-a1ØØ
Ø8F1:Ø1ØØ mov dx,1Ø9
Ø8F1:Ø1Ø3 mov ah,9
Ø8F1:Ø1Ø5 int 21
Ø8F1:Ø1Ø7 int 2Ø
Ø8F1:Ø1Ø9 db 'Good Morning, Robert!$'
Ø8F1:Ø11F
-
```

The "DB" Pseudo-Op

All the instructions in this program look pretty familiar except for this one:

```
Ø8F1:Ø1Ø9 db 'Good Morning, Robert!$'
```

This doesn't look like an ordinary assembly-language instruction, and it's not. In fact, it's a very strange sort of animal. Instead of being an instruction that tells the 8088 microprocessor to do something, it's an instruction that tells DEBUG (or the assembler program — when we get to that in the next chapter) what to do. In this case, it tells DEBUG to put all the bytes represented by the characters between the single quote marks into memory. Thus "G" is translated into its ASCII code 47h, "o" into 6Fh, and so on. These values are then placed in memory. Note that the "DB" itself is not placed in memory, since it is not really an instruction and is not going to be executed by the 8088. Once it has told

DEBUG to put the characters in memory, its job is done. It's called a "pseudo-op" because it's not really an "operation code" or instruction. It goes in the same place in the program as regular instructions, but it has a different purpose.

"DB" stands for "Define Byte," and as you can see it's very useful for putting ASCII codes into memory, since we don't have to look up the code for each value and then type it in with the "E" command. (If you don't have DOS version 2, you'll have to use the "E" command anyway, but you won't need to look up the values, since we'll show them when we disassemble the program with "U.")

You can also use "DB" to put numeric values into memory, either by themselves, or with ASCII characters. We'll show an example of this in the next section.

Don't forget the philosophical difference between regular assembler instructions like MOV and JMP (which are sometimes called "operation codes," or "op-codes") and pseudo-ops like DB. Instructions tell the *8088 microprocessor* what to do at the time the program is executed. Pseudo-ops, on the other hand, tell the *assembler program* (in this case DEBUG), what to do when the program is being assembled.

> "Instructions" are instructions to the microprocessor.
>
> "Pseudo-ops" are instructions to the assembler.

Here's the program unassembled (or disassembled) with "U":

```
-u100,108
0905:0100 BA0901        MOV     DX,0109
0905:0103 B409          MOV     AH,09
0905:0105 CD21          INT     21
0905:0107 CD20          INT     20
```

To see what bytes the db pseudo-op has placed in memory, "U" is not much help, since these bytes are not program instructions. Instead, we'll use "d," which provides not only the hex values of these bytes, but the ASCII characters they represent:

```
-d100,11f
08F1:0100  BA 09 01 B4 09 CD 21 CD-20 47 6F 6F 64 20 4D 6F   :..4.M!M Good Mo
08F1:0110  72 6E 69 6E 67 2C 20 52-6F 62 65 72 74 21 24 3A   rning, Robert!$:
                                                  Dollar sign
```

The program occupies the first 9 bytes of the top row. Then our string of characters starts, and continues all the way to the byte at 11E. Notice that the last byte is a dollar sign (24h), as required by the Print String function.

Save the program to your disk:

```
-nwakeup.com
-rbx
BX ØØØØ
:
-rcx
CX ØØØØ
:1F
-w
Writing ØØ1F bytes
```

Now, finally, run the program!

```
-g
Good Morning, Robert!
Program terminated normally
```

Terrific! It works just fine.

You can also execute the program directly from DOS, which can provide you with a nice way to start your day. Turn on the computer, and type:

```
A>wakeup
Good Morning, Robert!
```

It's kind of nice: a personal greeting from the cold impersonal machine. Of course, you can customize this program with your own name, simply by changing the phrase between the quotes in the "db" pseudo-op. Try it!

Buffered Keyboard Input Function

Now that you know how to print out a string of characters, how about reading a string in from the keyboard? Here's a DOS function which will do just that.

BUFFERED KEYBOARD INPUT Function — Number 0Ah

Enter with:

Reg AH = ØAh

Reg DS:DX = address of buffer

Execute:

```
INT 21
```

Return with: keyboard characters in buffer

Comments: First byte of buffer = maximum character count
Second byte = actual number of characters typed

Here's a program that makes use of the Buffered Keyboard Input DOS function.

```
A>debug
-a1ØØ
Ø8F1:Ø1ØØ mov dx,1Ø9
Ø8F1:Ø1Ø3 mov ah,a
Ø8F1:Ø1Ø5 int 21
Ø8F1:Ø1Ø7 int 2Ø
Ø8F1:Ø1Ø9 db 2Ø
Ø8F1:Ø1ØA
```

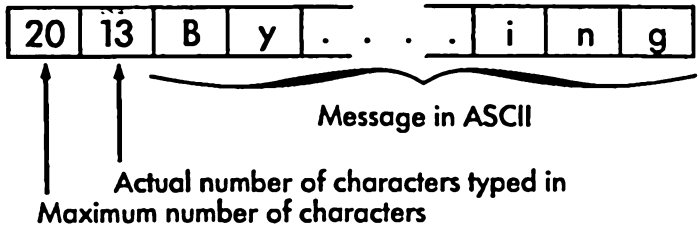
What's going on here? The idea is that the characters you type on the keyboard will be stored in a buffer (a buffer is just a sequence of memory locations). In the program above the buffer is defined in the program line

```
Ø8F1:Ø1Ø9 db 2Ø
```

This tells DEBUG to set aside 20h (32d) unused locations in memory. Your program tells the function where this buffer is by placing its address in the DX register with the MOV DX,109 instruction. The function number is Ah, so that's placed in the AH register, and then you call DOS with INT 21 as usual.

A "buffer" is a sequence of memory locations.

The key to using this function is to understand how the buffer is organized. Here's a diagram of the buffer:



The first byte of the buffer holds a number which is the maximum number of characters the function will accept from the keyboard — in this case 20h, which is 32d. If you type more characters than that, the beep sounds, and the cursor refuses to move any further right (as we will see shortly). This number can never be larger than 255d (FFH), since it only occupies one byte.

The second number is filled in by the function (not by your program) after you type in the message and press ↵. It's the actual number of characters you typed in. This is the only way your program can figure out how many characters were in your message.

The message itself goes in the bytes following these two one-byte numbers.

Let's try out our program and see what happens. If you want, save it to disk first:

```
-nbuffin.com
-rbx
BX ØØØØ
:
-rcx
CX ØØØØ
:a
-w
Writing ØØØA bytes
```

Let's examine it with "U" to make sure it looks right:

```
-ulØØ,1Ø8
Ø9Ø5:ØlØØ BAØ9Ø1        MOV     DX,Ø1Ø9
Ø9Ø5:Ø1Ø3 B4ØA          MOV     AH,ØA
Ø9Ø5:Ø1Ø5 CD21          INT     21
Ø9Ø5:Ø1Ø7 CD2Ø          INT     2Ø
```

To check that our buffer has been initialized properly we can dump just the single byte at 109 by typing:

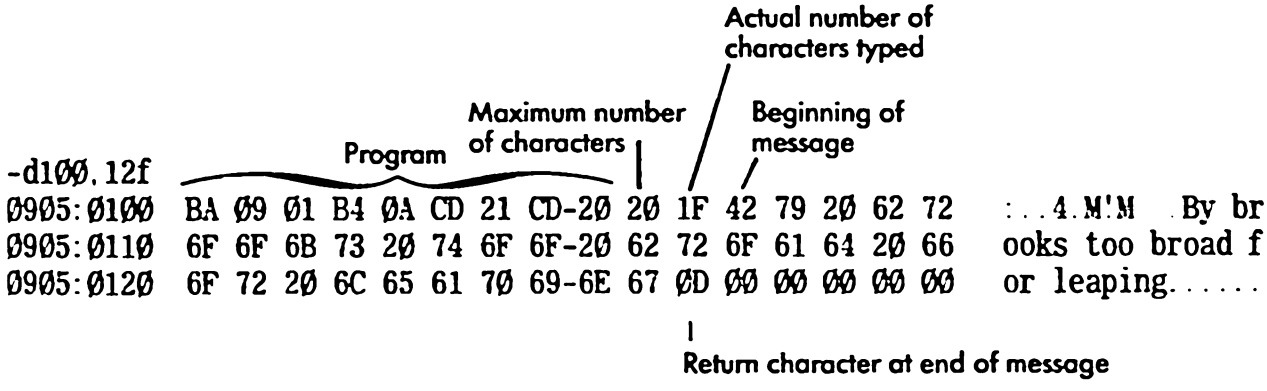
```
-d1Ø9,1Ø9
Ø9Ø5:Ø1Ø9  2Ø
```

The single byte at 109 is, of course, the maximum number of bytes the buffer can hold: 20h.

Let's run the program with "G", and then type something in:

```
-g
By brooks too broad for leaping
Program terminated normally    ↑
                               At this point we couldn't type any more characters.
```

Actually, we could only type 31d characters before the beeper sounded, not 32d, the number we put in the first byte of the buffer to specify the maximum number of characters (32d = 20h). This is because the enter character itself (often called a "return" or "carriage return" which has the value 0Dh) is always placed at the end of the message, and room needs to be saved for it. Look at the buffer with DEBUG:



If you run it again, but hit ↵ before you've filled up the buffer, then there will be a smaller count in the second byte of the buffer:

```
-g
Question Authority!
Program terminated normally
-d1ØØ,12f
Ø9Ø5:Ø1ØØ  BA Ø9 Ø1 B4 ØA CD 21 CD-2Ø 2Ø 13 51 75 65 73 74  :..4.M!M .Quest
Ø9Ø5:Ø11Ø  69 6F 6E 2Ø 41 75 74 68-6F 72 69 74 79 21 ØD 66  ion Authority!.f
Ø9Ø5:Ø12Ø  6F 72 2Ø 6C 65 61 7Ø 69-6E 67 ØD ØØ ØØ ØØ ØØ ØØ  or leaping.....
```

Actual number of characters typed

Notice how the remnant of the old message is left in the buffer. We (or our program) know that this is junk because the character count is only 13h, which is 19d: the number of characters in "Question Authority!"

The Mirror Program

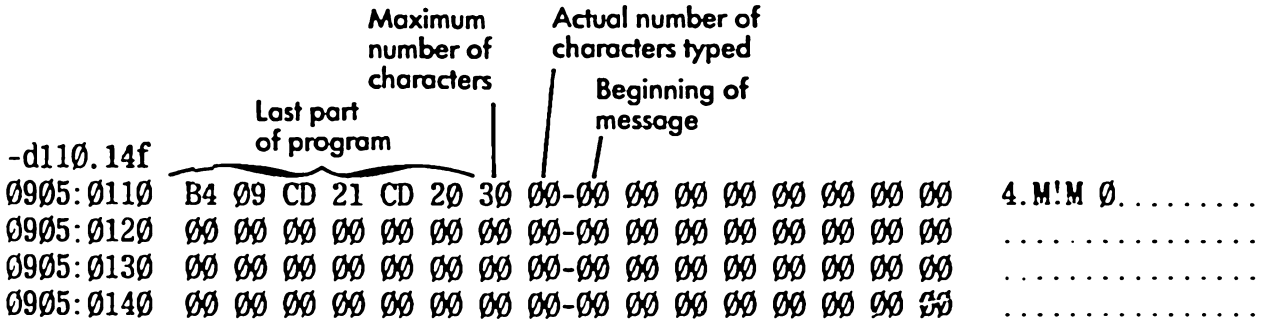
Let's put together the last two DOS functions we've learned, and make a program that takes a sentence you type in, and echos or "mirrors" it back onto the screen.

```
-a1ØØ
Ø9Ø5:Ø1ØØ mov dx,116
Ø9Ø5:Ø1Ø3 mov ah,a      <— Buffered Keyboard Input
Ø9Ø5:Ø1Ø5 int 21
Ø9Ø5:Ø1Ø7 mov dl,a
Ø9Ø5:Ø1Ø9 mov ah,2      <— Display Output (linefeed)
Ø9Ø5:Ø1ØB int 21
Ø9Ø5:Ø1ØD mov dx,118
Ø9Ø5:Ø11Ø mov ah,9      <— Print String
Ø9Ø5:Ø112 int 21
Ø9Ø5:Ø114 int 2Ø        <— Return to DEBUG or DOS
Ø9Ø5:Ø116 db 3Ø         <— Maximum characters
Ø9Ø5:Ø117
```

Type this in (don't type in the comments on the right, of course). Then save it to disk in the usual way as MIRROR.COM. Here's what it looks like with "U":

```
-u1ØØ,115
Ø9Ø5:Ø1ØØ BA16Ø1        MOV     DX,Ø116
Ø9Ø5:Ø1Ø3 B4ØA          MOV     AH,ØA
Ø9Ø5:Ø1Ø5 CD21          INT     21
Ø9Ø5:Ø1Ø7 B2ØA          MOV     DL,ØA
Ø9Ø5:Ø1Ø9 B4Ø2          MOV     AH,Ø2
Ø9Ø5:Ø1ØB CD21          INT     21
Ø9Ø5:Ø1ØD BA18Ø1        MOV     DX,Ø118
Ø9Ø5:Ø11Ø B4Ø9          MOV     AH,Ø9
Ø9Ø5:Ø112 CD21          INT     21
Ø9Ø5:Ø114 CD2Ø          INT     2Ø
```

And here's the buffer, which starts at 116:



Let's see what the program does. As you can see, it's divided into three major parts. The first part gets our input from the keyboard and puts it into the buffer set up at location 116. The message actually goes into the addresses starting at 118; as before, 116 holds the maximum count, and 117 is filled in with the actual number of characters received from the keyboard.

The second part of the program uses the Display Output function to print a linefeed. (The ASCII code for a linefeed is 0Ah, which we type in as "a". This is necessary because Buffered Keyboard Input prints a carriage return at the end of the string, but not a linefeed; so that the cursor simply goes back to the beginning of the line you typed in. If we started printing here, we would write directly over what we had just typed in. Yes, we forgot this when we first wrote the program!)

Finally, the third part of the program uses the Print String function to print out the phrase we typed in. As far as this function is concerned, the buffer starts at the place where the first character of the message is, not where the max count is, so 118 is the number we put in the DX register.

Another important thing to remember about the Print String function is that the only way it knows when to stop outputting characters is when it sees the dollar sign ($). It's very important, therefore, *to terminate the sentence you type in with a "$" sign.* Otherwise this function will run amok, madly printing all the junk it finds in memory, and covering the screen with weird symbols. A more sophisticated program could fill in the dollar sign for us, but this simple program is not fail-safe in this regard.

All right, let's try it out. Start the program with "g", and then type something in:

```
-g
The curfew tolls the knell of parting day.$     <— You type this
The curfew tolls the knell of parting day.      <— Program prints this
Program terminated normally
-
```

There it is, a perfect echo.

Now you can look at the buffer with "d" to see what happened:

```
-d11Ø,14f
Ø9Ø5:Ø11Ø  B4 Ø9 CD 21 CD 2Ø 3Ø 2B-54 68 65 2Ø 63 75 72 66   4.M!M Ø+The curf
Ø9Ø5:Ø12Ø  65 77 2Ø 74 6F 6C 6C 73-2Ø 74 68 65 2Ø 6B 6E 65   ew tolls the kne
Ø9Ø5:Ø13Ø  6C 6C 2Ø 6F 66 2Ø 7Ø 61-72 74 69 6E 67 2Ø 64 61   ll of parting da
Ø9Ø5:Ø14Ø  79 2C 24 ØD ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   y,$.............
-q                  └──────── This is the obligatory dollar sign ────────┘
```

One of the important things to notice in this program is the different way the two function calls deal with the buffer. The address passed to Buffered Keyboard Input in the DX register is actually two bytes before the start of the string, to allow for the "maximum-characters" number and the "characters-typed-in" number. The number passed to Print String, however, is at the start of the string itself. Since Print String pays no attention to the "characters-typed-in" number, a "$" must be used to end the string. Figure 4-4 shows how the buffer looks to the two functions.

Writing to the Printer

Let's see if we can get the hang of using a whole new piece of peripheral equipment, the printer. We're also going to talk about an important new idea called *indirect addressing*, so even if you don't have a printer you should read this section.

The Printer Output Function

PRINTER OUTPUT Function — Number 05h

Enter with:

Reg AH = 5

Reg DL = character to be printed

Execute:

```
INT 21
```

Characters are sent to the printer using the Printer Output function which sends *one character at a time* from the DL register, much as the Display Output function does for the video screen. (In fact, if you don't have a printer you can rewrite these programs to work with the Display Output function, so that you can see what they do.) There is no function which sends a string of characters to the printer all at once, as the Print String function does to the video.

This leads to an interesting problem. Suppose we have a string of characters in memory somewhere, such as we did in the last section when we filled up a buffer in memory with characters using the Buffered Keyboard Input function. And suppose we want to send these characters to the printer to be printed out. How do we get the characters from their

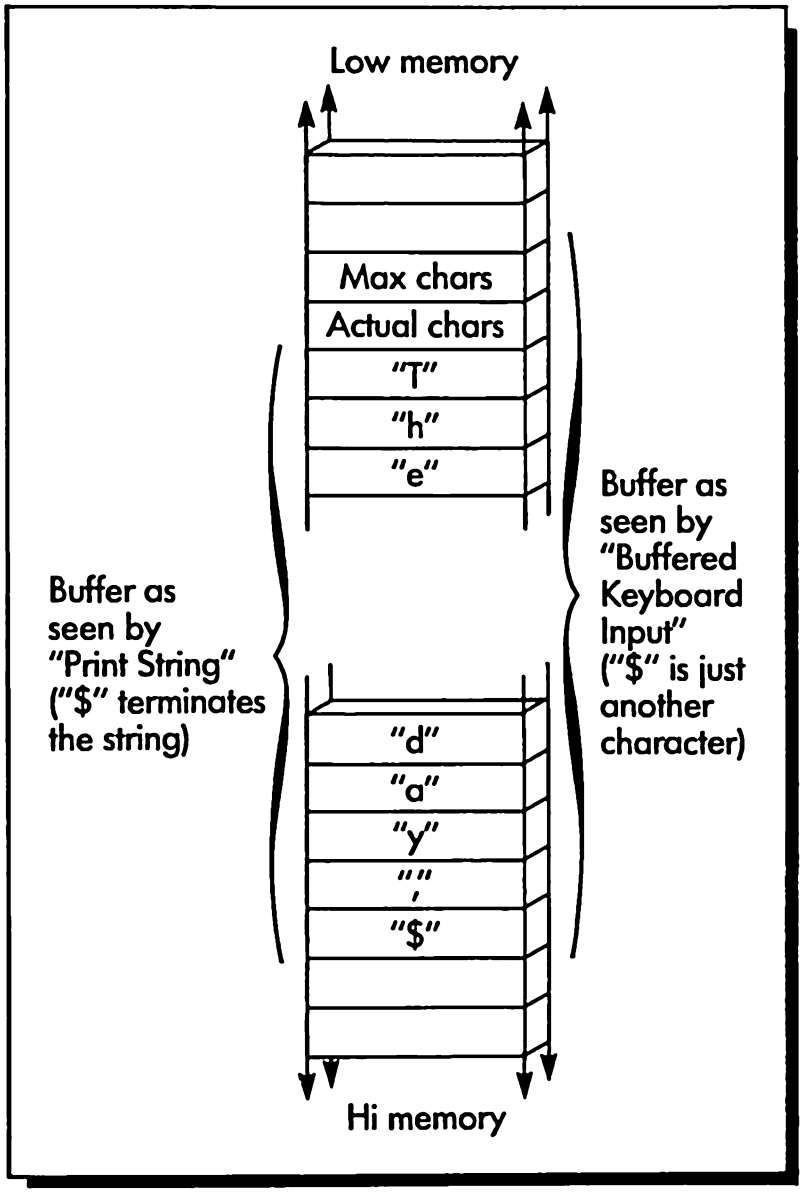


Figure 4-4. How the buffer is used in the MIRROR program

memory locations into the DL register so we can send them to the printer?

We're going to show you two different ways to access strings of characters in memory. The first is clumsy, but will serve as an introduction to a technique called *indirect addressing*. The second will make use of indirect addressing in a different way, to do the job considerably more elegantly.

The Not-So-Elegant Way to Print a String

In the following program we're going to send the word "hi" to the printer. Then, since the printer will not actually print anything unless it has received an entire line (80 characters), or unless it receives a carriage return, we'll send it a carriage return. Finally, to make sure that the next line we print doesn't print over the first one, we'll send it a linefeed as well. These four characters — "h", "i", 0Dh (the ASCII code for a carriage return), and 0Ah (the ASCII code for a linefeed) — will all be assembled by DEBUG into a buffer in memory.

Type in the program shown below. (The square brackets, which you haven't encountered before, are on the two keys to the right of the "P" key, and are lowercase. We'll explain in a moment what they do.)

```
-a1ØØ
Ø9Ø5:Ø1ØØ mov dl,[122]      <— Send "h" to printer
Ø9Ø5:Ø1Ø4 mov ah,5
Ø9Ø5:Ø1Ø6 int 21
Ø9Ø5:Ø1Ø8 mov dl,[123]      <— Send "i" to printer
Ø9Ø5:Ø1ØC mov ah,5
Ø9Ø5:Ø1ØE int 21
Ø9Ø5:Ø11Ø mov dl,[124]      <— Send carriage return to printer
Ø9Ø5:Ø114 mov ah,5
Ø9Ø5:Ø116 int 21
Ø9Ø5:Ø118 mov dl,[125]      <— Send linefeed to printer
Ø9Ø5:Ø11C mov ah,5
Ø9Ø5:Ø11E int 21
Ø9Ø5:Ø12Ø int 2Ø            <— Return
Ø9Ø5:Ø122 db 'hi',Ød,Øa     <— ASCII string
Ø9Ø5:Ø126
```

Save the program if you want, as "PRINTHI.COM".

```
-rbx
BX ØØØØ
-rcx
CX ØØ1E
```

```
:26
-nprinthi.com
-w
Writing ØØ26 bytes
```

Use "U" to see that you typed it in accurately:

```
-u1ØØ,121
Ø9Ø5:Ø1ØØ 8A1622Ø1      MOV     DL,[Ø122]
Ø9Ø5:Ø1Ø4 B4Ø5          MOV     AH,Ø5
Ø9Ø5:Ø1Ø6 CD21          INT     21
Ø9Ø5:Ø1Ø8 8A1623Ø1      MOV     DL,[Ø123]
Ø9Ø5:Ø1ØC B4Ø5          MOV     AH,Ø5
Ø9Ø5:Ø1ØE CD21          INT     21
Ø9Ø5:Ø11Ø 8A1624Ø1      MOV     DL,[Ø124]
Ø9Ø5:Ø114 B4Ø5          MOV     AH,Ø5
Ø9Ø5:Ø116 CD21          INT     21
Ø9Ø5:Ø118 8A1625Ø1      MOV     DL,[Ø125]
Ø9Ø5:Ø11C B4Ø5          MOV     AH,Ø5
Ø9Ø5:Ø11E CD21          INT     21
Ø9Ø5:Ø12Ø CD2Ø          INT     2Ø
```

And "d" to look at the ASCII string:

```
-d12Ø,12f
Ø8F1:Ø12Ø  6C 46 68 69 ØD ØA 87 DA-E8 86 FØ E8 DE FA ØE ØØ   lFhi...Zh.phˆz..
```

ASCII values

- 6C = "h"
- 68 = "i"
- 69 = Carriage return
- ØD = Linefeed
- ØA

Now, run the program, making sure that your printer is turned on, and set to "on-line":

```
-g                                  ← Nothing printed on screen, but printer prints
Program Terminated Normally
```

If all went well, your printer printed the word "hi." If you run the program a second time, it will print "hi" again on the next line, and so on.

Indirect Addressing

Now we're going to talk about the square brackets in the program.

Recall that if you have the instruction

```
mov dl,122
```

the number 122h will be placed in the DL register. What about the same instruction with brackets around the 122?

```
mov dl,[122]
```

This instruction takes *the contents of memory location 122* and places it in register DL. If you aren't familiar with this concept of *indirect addressing* it may take a little getting used to.

Look at the dump we made of the ASCII string in locations 122 to 125. The first letter, "h," is in location 122. When the instruction

```
mov dl,[122]
```

is executed, the *contents* of location 122, which is 68h (the ASCII code for "h"), will be placed in the DL register. The number 122 itself doesn't go anywhere, but the instruction *uses it* to figure out where to get the "h". Figure 4-5 shows how this works.

To print the next character we execute the same sequence of instructions, but this time we take the character out of memory location 123. This is the "i" character. We do the same thing with the carriage return (0Dh), and linefeed (0Ah), and our message is completed.

However, you will no doubt have noted how inefficient this program

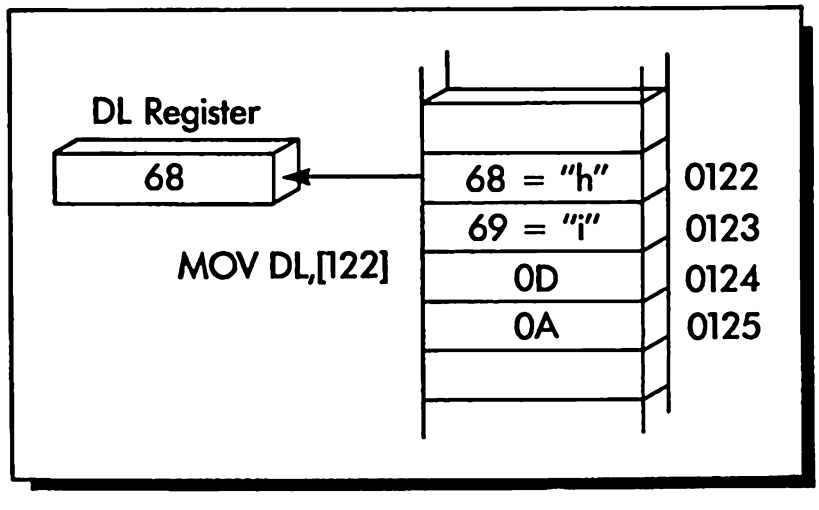


Figure 4-5. Operation of the MOV DL,[122] instruction

is. It has to execute a completely different instruction every time it wants to load a different character into the DL register. If you're thinking that there must be a better way, you're absolutely right. We've introduced you to the idea of indirect addressing in its simplest form; now let's go on and see how powerful an idea it is when it's used in a slightly different way.

> "Indirect addressing" means referring to the address of something, rather than to the something.

A More Elegant Way to Print a String

Use DEBUG to type the following program into memory. This program will send a complete string of any length to the printer. In this case the string (courtesy of an anonymous 16th century poet) is in line 111:

```
-a1ØØ
Ø9Ø5:Ø1ØØ mov cx,31          <— Number of characters to print
Ø9Ø5:Ø1Ø3 mov bx,111         <— Address of first character
Ø9Ø5:Ø1Ø6 mov dl,[bx]        <— Put the character in DL
Ø9Ø5:Ø1Ø8 mov ah,5           <— Printer Output DOS function
Ø9Ø5:Ø1ØA int 21             <— Call DOS
Ø9Ø5:Ø1ØC inc bx             <— Increment the pointer
Ø9Ø5:Ø1ØD loop 1Ø6           <— Loop until done
Ø9Ø5:Ø1ØF int 2Ø             <— Return to DEBUG or DOS
Ø9Ø5:Ø111 db 'She is most fair, though she be marble-hearted.',Ød,Øa
Ø9Ø5:Ø142
```

Here's the "U" listing:

```
-u1ØØ,1Øf
Ø9Ø5:Ø1ØØ B931ØØ        MOV     CX,ØØ31
Ø9Ø5:Ø1Ø3 BB11Ø1        MOV     BX,Ø111
Ø9Ø5:Ø1Ø6 8A17          MOV     DL,[BX]
Ø9Ø5:Ø1Ø8 B4Ø5          MOV     AH,Ø5
Ø9Ø5:Ø1ØA CD21          INT     21
Ø9Ø5:Ø1ØC 43            INC     BX
Ø9Ø5:Ø1ØD E2F7          LOOP    Ø1Ø6
Ø9Ø5:Ø1ØF CD2Ø          INT     2Ø
```

You can save this program on your disk as "PSTRING.COM". Here's what the string itself looks like in memory, using "D":

```
-d111,141
09Ø5:Ø111  53 68 65 2Ø 69 73 2Ø-6D 6F 73 74 2Ø 66 61 69      She is most fai
09Ø5:Ø12Ø  72 2C 2Ø 74 68 6F 75 67-68 2Ø 73 68 65 2Ø 62 65  r, though she be
09Ø5:Ø13Ø  2Ø 6D 61 72 62 6C 65 2D-68 65 61 72 74 65 64 2E   marble-hearted.
09Ø5:Ø14Ø  ØD ØA
```

Now you can run the program in the usual way with "G" and see the entire line of poetry printed out. Notice how short the program is, and how long a string it can print. In fact, the string can be as long as you want. How does this program work?

Indirect Addressing with a Register Pointer

In the PSTRING program the square brackets do not surround a memory address as they did earlier in the PRINTHI program. Instead they surround *the BX register*:

```
09Ø5:Ø1Ø6 8A17          MOV     DL,[BX]
```

This instruction makes use of a special property of the BX register: If you put a memory address in BX, and then use BX with brackets around it in an instruction, references to [bx] will operate *on the contents of the memory address contained in BX, not on BX itself.*

Let's see how this fits into the program. Most of the program is enclosed in a loop, as we can see from the MOV CX,31 instruction at the beginning of the program, and the LOOP 106 instruction near the end. We put the number of characters to be printed into CX so the loop will be executed this many times, then load the address of the first character into BX so that BX will *point to* this character. By "point to" we mean that BX now contains the memory address of the character. Now, using the square brackets, we write the instruction MOV DL,[BX], which means: "Take the character which is in the memory location which is in BX, and put this character in DL." Figure 4-6 shows how this works.

Now comes the clever part, where we use the real power of indirect addressing. Instead of having to write another instruction to get the *next* character in the string into DL (as we did in the last example), all we have to do is *increment the address in the BX register*, and then execute the same MOV DL,[BX] instruction again. Since the address in BX will now be 112 instead of 111, the character we put into DL will be the "h" in "She" instead of the "S." We can proceed like this through the entire string, printing the characters until CX reaches 0, and the LOOP instruction no longer causes a jump back to the start of the loop, but instead "falls through" (goes on to) the INT 20 instruction which ends the program.

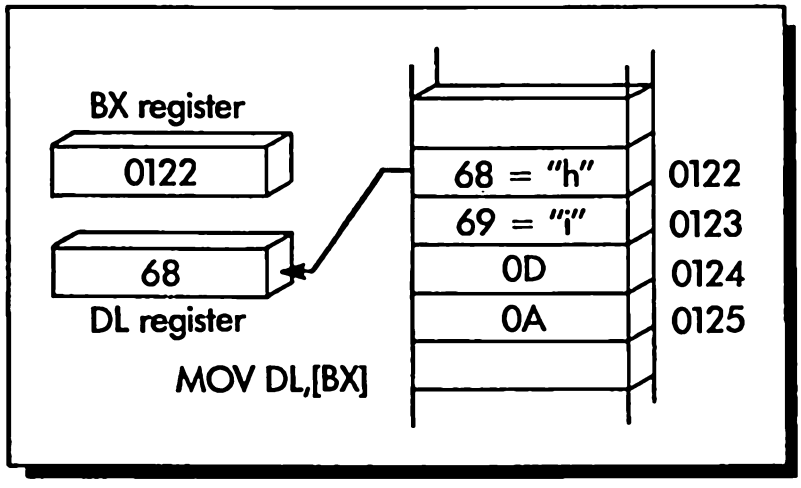


Figure 4-6. Operation of the MOV DL,[BX] instruction

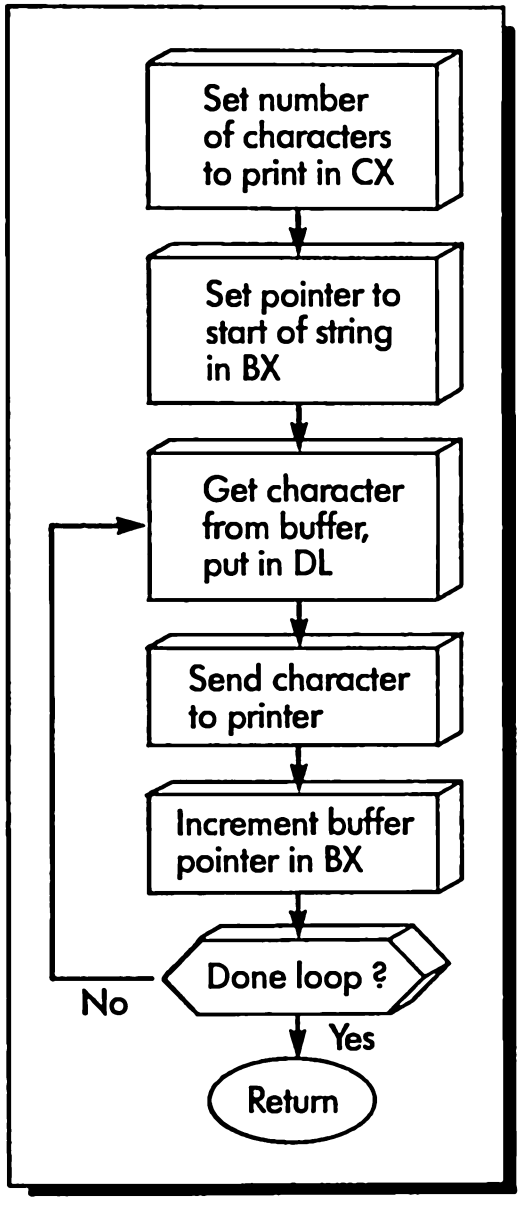


Figure 4-7. Flow chart of the PSTRING program

Figure 4-7 shows a flow chart of the operation of the PSTRING program.

We'll be dealing with indirect addressing from time to time in the following chapters, so you'll have a chance to see it in action again, if it wasn't entirely clear to you this time around.

Remember the PSTRING program. At some point, when you want to print something out on the printer, you'll probably need to turn this program into a little subroutine that can be used in larger programs to provide printer output.

Sending Control Codes to Your Printer

While we're on the subject of the printer, let's talk about how to send it control codes. What are control codes? They're one- two- or three-character codes that tell the printer to do things like print condensed characters, double width characters, or emphasized characters; or to change the number of lines per inch or the number of characters per line; or to skip over perforations at the end of a page. There are dozens of these control codes, and without assembly language there is really no easy way to send them to the printer. It can be done in BASIC, but that means loading BASIC every time you want to change something in the printer, which can be an inconvenience.

This discussion is geared to the standard IBM dot-matrix printer, which is in reality an Epson MX-80. If you are using another kind of printer the control codes may be different, but the principles are the same. You'll simply have to look up the codes in the manual that comes with your printer, and apply them as in the following examples.

MODE, a DOS command, can be used to change the characters per line and the lines per inch vertical spacing on the printer, so we don't need to do that. Instead, let's write a little routine to shift into "emphasized" printing. In this mode, the print head makes two passes across the page, with the paper rotated very slightly in between passes. The result is that the spaces between the dots are filled in, resulting in a much better looking print, which is sometimes called "correspondence quality" (a more or less appropriate term, depending on who the correspondence is to).

Turning On Emphasized Print

The control code that we've used for emphasized print is <escape>"E". You'll need to check the manual for your printer to find your code. In BASIC you would send this code to the printer with the statement

```
LPRINT CHR$(27)"E"
```

27d is the ASCII code for the "escape" character.

Here's how to do it in assembly language:

```
-alØØ
Ø9Ø5:ØlØØ mov cx,2
Ø9Ø5:ØlØ3 mov bx,111
Ø9Ø5:ØlØ6 mov dl,[bx]
Ø9Ø5:ØlØ8 mov ah,5
Ø9Ø5:ØlØA int 21
Ø9Ø5:ØlØC inc bx
Ø9Ø5:ØlØD loop lØ6
Ø9Ø5:ØlØF int 2Ø
Ø9Ø5:Ø111 db 1b,'E'
Ø9Ø5:Ø113
```

This is exactly the same program as the one in the last section which sent a string to the printer. This version also sends a string to the printer, but in this case the string, in locations 111 and 112, consists of only two characters: a 1Bh, which is the ASCII code for the "escape" character, and an "E", which is the specific character that puts the printer into emphasized mode. Thus we put a 2 into CX at the beginning of the program, since we only want to print two characters.

The "U" listing of the program is:

```
-ulØØ,lØf
Ø9Ø5:ØlØØ B9Ø2ØØ        MOV     CX,ØØØ2
Ø9Ø5:ØlØ3 BBllØl        MOV     BX,Ø111
Ø9Ø5:ØlØ6 8A17          MOV     DL,[BX]
Ø9Ø5:ØlØ8 B4Ø5          MOV     AH,Ø5
Ø9Ø5:ØlØA CD21          INT     21
Ø9Ø5:ØlØC 43            INC     BX
Ø9Ø5:ØlØD E2F7          LOOP    Ø1Ø6
Ø9Ø5:ØlØF CD2Ø          INT     2Ø
```

And examining the two characters with "d" shows:

```
-dllØ,llf
Ø8F1:Øll0  76 1B 45 CØ A2 7Ø 46 A2-6F 46 A2 6E 46 Ø4 Ø2 A2   v.E@"pF"oF"nF.."
```

"Escape" code

Letter "E"

ASCII equivalents

So, try out the program! You can either type "G" from DEBUG, or you can save the program in the usual way as "EMPHAP.COM" (for

"EMPHAsized Print"), and execute it from DOS — the obvious course of action if you want to be able to use the program conveniently whenever you want.

When you run the program the printer may or may not make a little clicking sound. In either case it should put itself into "emphasized mode." All subsequent characters sent to it will be printed in emphasized print, as you can verify by typing Ctrl Prtsc to turn on the printer, and then typing something.

Wow! Look at that! If you haven't used emphasized mode before, you'll be amazed at how nice it is. Of course it takes twice as long to print everything, because the printing head has to make two passes across the page for every line; but this is a small price to pay if you are writing a letter that may land you a hot new job.

Turning Off Emphasized Print

How do you get back to normal print mode, when you no longer want emphasized print? Simply send another control code, this time <escape>"F". Here are the "A" and "U" listings for a program to do that. As you can see, it is identical to the EMPHAP program, except that it sends <escape>"F" instead of <escape>"E".

```
-a1ØØ
Ø9Ø5:Ø1ØØ mov cx,2
Ø9Ø5:Ø1Ø3 mov bx,111
Ø9Ø5:Ø1Ø6 mov dl,[bx]
Ø9Ø5:Ø1Ø8 mov ah,5
Ø9Ø5:Ø1ØA int 21
Ø9Ø5:Ø1ØC inc bx
Ø9Ø5:Ø1ØD loop 1Ø6
Ø9Ø5:Ø1ØF int 2Ø
Ø9Ø5:Ø111 db 1b,'F'
Ø9Ø5:Ø113

-u1ØØ,1Øf
Ø9Ø5:Ø1ØØ B9Ø2ØØ        MOV     CX,ØØØ2
Ø9Ø5:Ø1Ø3 BB11Ø1        MOV     BX,Ø111
Ø9Ø5:Ø1Ø6 8A17          MOV     DL,[BX]
Ø9Ø5:Ø1Ø8 B4Ø5          MOV     AH,Ø5
Ø9Ø5:Ø1ØA CD21          INT     21
Ø9Ø5:Ø1ØC 43            INC     BX
Ø9Ø5:Ø1ØD E2F7          LOOP    Ø1Ø6
Ø9Ø5:Ø1ØF CD2Ø          INT     2Ø
```

Here too is the "D" dump showing the two characters in the message buffer:

```
-d11Ø,11f
Ø8F1:Ø11Ø  76 1B 46 CØ A2 7Ø 46 A2-6F 46 A2 6E 46 Ø4 Ø2 A2   v.F@"pF"oF"nF.."
             |  └ letter "F"                                    Y
             └── "Escape" code                                  ASCII equivalents
```

Save this program as "NORMALP.COM" (for NORMAL Print). Try it out. The printer should revert to normal mode.

You can modify these little programs to do other things to your printer. Sometimes you need to send it only one control character (instead of the two in these examples), and sometimes it needs three. For example, for double-width printing you need to send three characters: <escape>"W"1 (that's "W" followed by the number 1). <escape>"W"0 turns double-width mode off. <escape>"A"18h causes the printer to double space.

Now that you know how to take command of the printer, you could be in great demand by IBM PC users who want a quick way to tell their printer to change modes, and are tired of doing it in BASIC, or not at all.

Summary

In this chapter you've practiced your assembly-language skills at the same time you learned about DOS function calls. You've been introduced to the Disk Operating System and its various parts, and seen how they work together. You should know what function calls are, and how they are typically called from your assembly-language program. You've learned a number of new function calls, and you know how to get strings from the keyboard and send them to the video screen and the printer.

We've also covered indirect addressing — a powerful technique for accessing memory — and its use with the BX register as a pointer. You may be surprised that we have introduced no new 8088 instructions. In fact, the ones you've learned already are so powerful, when used with function calls, that you've needed only five instructions for all the programs in this chapter.

In the next chapter you'll learn some new instructions, and also how to use MASM and ASM, the full-size, ultra-sophisticated assemblers we'll be using in the rest of the book.

5

Introduction to the IBM MACRO Assembler

Concepts

- MASM and ASM
- Source and object files
- TXT, OBJ, EXE, and COM files
- The LINK and EXE2BIN programs
- LST files
- Deciphering machine language op-codes
- Using batch processing

In this chapter we're going to move into the big leagues and introduce you to the MACRO assembler that's available from IBM as the standard assembler program for the PC. We'll talk about the purpose of an assembler, its differences from the "A" command in DEBUG, and then use the assembler to assemble the happy face program that was your first programming example in chapter 2. You'll learn how to create a "source file" using your word processor or text editor and how to assemble, link, and convert your program to a COM "object file."

For practice, we'll run through the process again with a second program you're already familiar with, the SMASCII program from chapter 4, and show you how to use another important feature of the assembler, the LST file, which gives the most complete possible picture of your program.

This chapter will use short examples to explain the operation of the assembler. In the next chapter you'll put what you've learned to more serious use.

MASM and ASM

As we mentioned in the Introduction, IBM actually provides two different assemblers on the same disk when you buy the IBM MACRO Assembler. One is MASM, the full-scale version of the assembler. The second is ASM, a small-scale version of MASM which takes up less memory and therefore loads from the disk somewhat faster than MASM, but which lacks some of MASM's more sophisticated features. The difference in memory is important if you have a 64K machine, since only ASM will fit in this space. To run MASM you need at least 96K; preferably more, so you can assemble larger programs. The features left out of ASM are macros, conditional assembly, and full printout of error messages.

If you have only 64K you will have to use ASM. If you have 96K or more you have a choice. Our preference is to use ASM, since it is smaller and therefore loads from the disk into the PC somewhat faster than MASM. Since we will be doing a great many comparatively short assemblies in this book, it makes sense to use the fastest possible method. The error messages — which are given as numbers in ASM — are easily looked up in the back of the *IBM Personal Computer MACRO Assembler.*

However, whether to use ASM or MASM is really a matter of personal preference (that is, if you have 96K or more). Try out both assemblers. They will operate in just the same way, except for the points noted above, so either can be used with this book. Since we prefer to use ASM, we will refer to ASM when we are talking about the assembler, but you can use MASM instead if you wish.

Word Processor, Text Editor, or Programs

As we mentioned in the introduction, you will need some sort of text editor or word processor program in order to create the text files that constitute the input to the assembler. It is possible to use EDLIN, the text editor included as part of the PC-DOS operating system. This works adequately, especially on short programs. However, as the programs get longer you will appreciate more and more some of the advanced features of a good word processing program, such as screen editing (moving the cursor around the screen to make corrections, rather than doing it line by line as in EDLIN), the ability to move quickly from one part of a long document to another, the greater ease of making insertions and deletions in a line, and the ability to move entire blocks of text from one part of the document to another.

We are going to assume that you have either EDLIN or a word

processor up and running on your system, and that you know enough about how to use it that you can type in the simple example programs from this book.

If you are using a word processor that has a "nondocument" mode — that is, one designed especially for program listings (as opposed to paragraphs of prose) — then you should use that feature.

What Does an Assembler Do?

You already know — from using the "A" command in DEBUG — that an assembler takes symbolic instructions (like MOV AH,2) and transforms them into machine language: the binary numbers which are actually placed in the memory of the computer to be executed by the 8088 microprocessor (like 10110100 00000010). We see a hexadecimal representation of these binary numbers (B402) when we use the "U" command in DEBUG to "unassemble" our program. This process is shown in Figure 5-1.

As you learned, a program consisting of binary numbers can then be saved on the disk as a COM file.

Input to the Assembler

The operation of a true assembler such as ASM (or MASM) is somewhat different from that of the DEBUG "A" command. The first difference is that instead of typing the symbolic instructions directly into the program, as you do in DEBUG, you type them into a completely

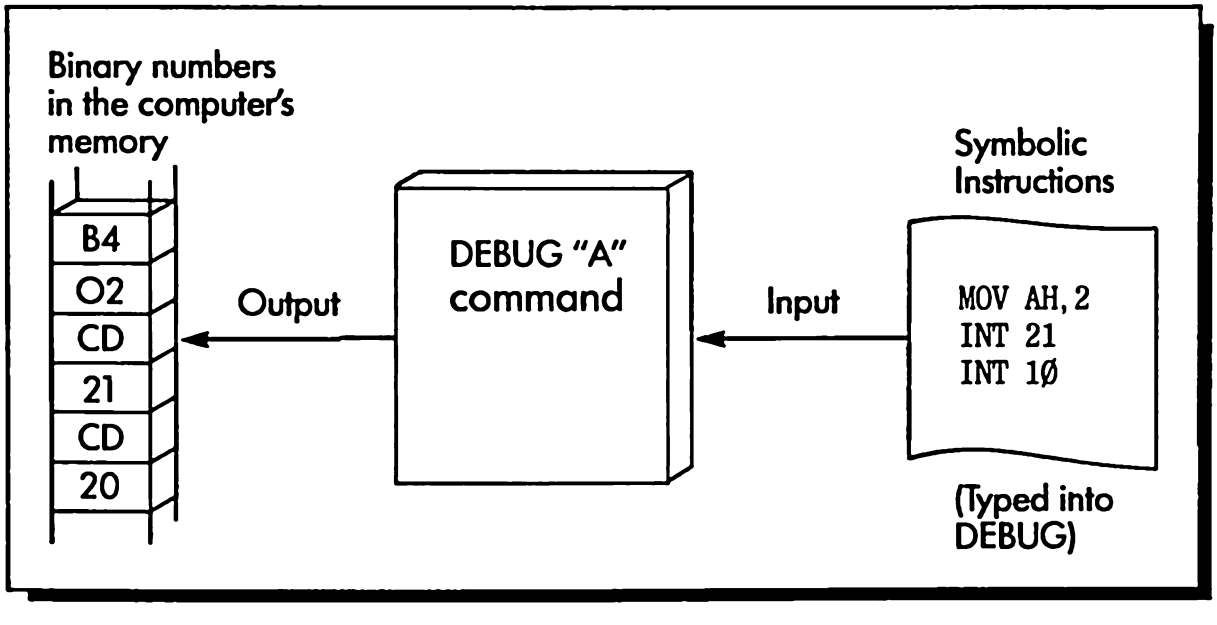


Figure 5-1. Input and output of the DEBUG "A" command

separate *text file* created with a word processing program. This text file — which must always have the file extension ASM (as in MYPROG.ASM) — then becomes the input for the ASM program.

> Source files for ASM (or MASM) always have the file extension ASM

Output from the Assembler

The *outputs* of the "A" command and ASM are also different. The "A" command simply puts binary numbers into the computer's memory, in the form of a program which can be immediately executed. ASM, on the other hand, creates a disk file called an OBJ (for "OBJect") file. This OBJ file cannot simply be loaded into memory and executed. It is in a complex format and contains information about where various parts of the program are to be loaded and how they might be combined with other programs.

Figure 5-2 shows the input and output of the ASM program.

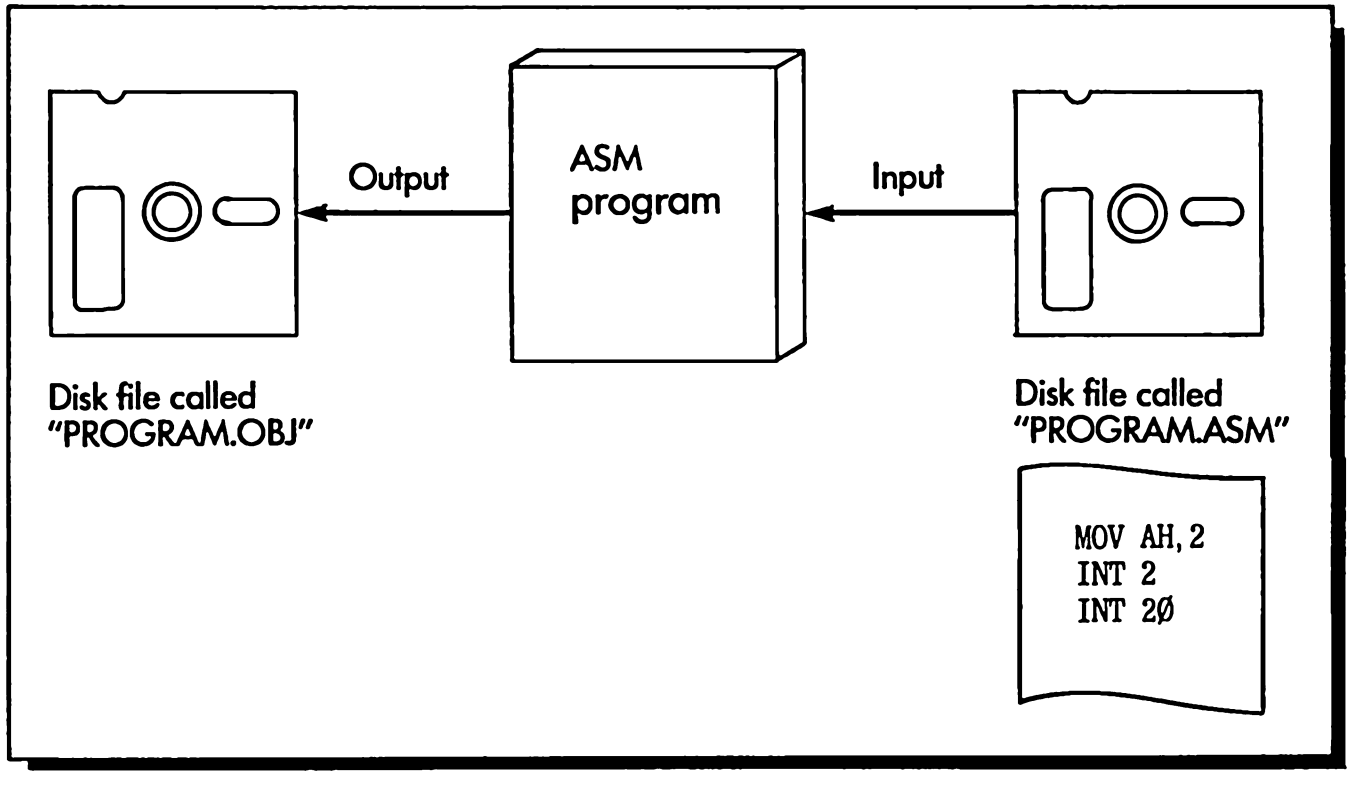


Figure 5-2. Input and output of ASM

The LINK and EXE2BIN Programs

It's the job of a completely different program, called LINK, to turn the OBJ file into a file which can be loaded into memory and executed. At this point we don't need to know too much about what LINK does. (We'll cover the more sophisticated uses of LINK in later chapters.) We do need to know that the output of LINK is a kind of file called an EXE (for "EXEcute") file. EXE files can be loaded into memory directly and executed. They can also, if they are written correctly, be converted into COM files which are just the same as the COM files that result from saving to disk a program typed into DEBUG using the "A" command. A program called EXE2BIN ("EXE to BINary") is used for this conversion. (Actually EXE2BIN gives its output file — unless it's told otherwise — an extension of BIN, but this is essentially the same as COM. More on this later.)

What a lot of file extensions to have to think about! Don't worry. It's not really as bad as it looks. Figure 5-3 shows the relationship of the programs ASM, LINK, and EXE2BIN to ASM, OBJ, EXE and COM files.

COM or EXE, What's the Difference?

Both COM and EXE files can be loaded into memory and executed as programs. What's the difference between these two kinds of files, and why would we prefer one over another?

A COM file is the simplest way of storing a program. It consists only of the binary numbers that make up the program. Since there is no other information in the file besides these binary numbers, a COM file takes up the smallest possible amount of space in memory; whereas an EXE file, which also contains a "header" consisting of various information about the file, is much longer. If your program is 100 bytes long, the COM file will be 100 bytes long. An EXE file, on the other hand, has a *minimum* length of 640 bytes, even if your program is only 2 bytes long. Because they are smaller and simpler, COM files load faster than EXE files.

Another advantage of COM files for our purposes in this chapter is that they can make use of the simple INT 20 return to DEBUG. INT 20 works all right in EXE programs executed directly from DOS, but in DEBUG it causes trouble, and we'll be using DEBUG a good bit in this chapter to execute and interact with our programs. EXE files require a more complex return procedure, so we can avoid a little overhead by using COM files.

On the other hand, there are disadvantages with COM files. First, they cannot occupy more than 64K of memory space. This is not a

problem in the relatively small programs we will be writing in this book, but for large system programs it can be a disadvantage.

Second, COM files cannot be linked up with other files when they are

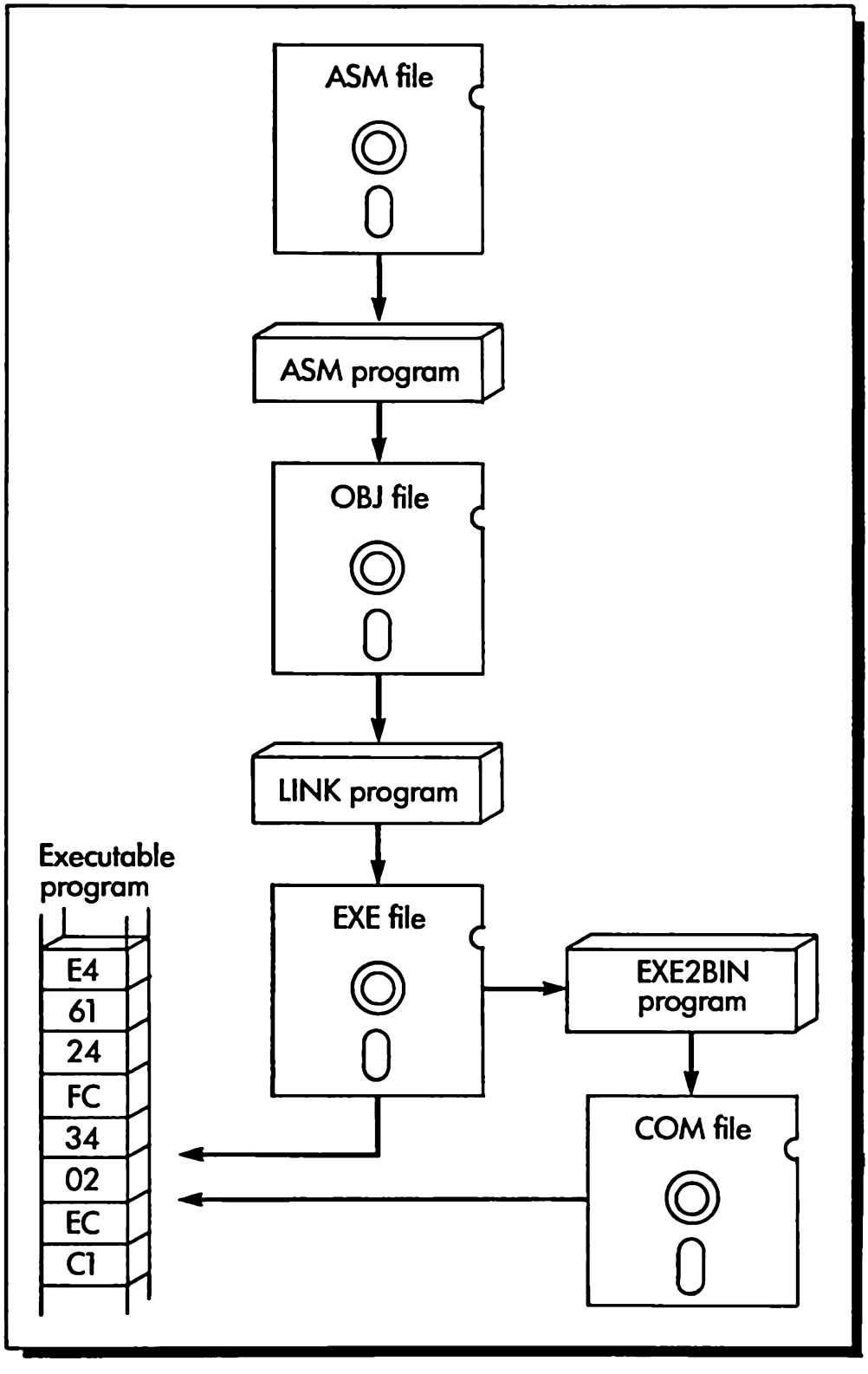


Figure 5-3. Relationship of ASM, LINK, and EXE2BIN

loaded. The real purpose of the LINK program is to be able to combine several different programs into a larger program when they are loaded. This can have all sorts of advantages in sophisticated programming situations. We'll cover some of these toward the end of the book, but for the moment we don't really care about this advantage of EXE files either.

The third advantage of EXE files is that it is easier to make use of different memory segments to perform different functions. We'll be discussing memory segmentation in a later chapter, so again, for the time being, this advantage is not important to us.

The last advantage of EXE files *is* important to us, and that is the fact that it takes one more step to turn an ASM file into a COM file than it does to turn it into an EXE file: the use of the EXE2BIN program to change the EXE file into a COM file. This is an inconvenience, but not a serious one for short programs.

All things considered, it seems easiest to start off producing COM files, so that's what we'll do in the balance of this chapter.

Assembling Your First Program

Now that you have a rough idea of what an assembler is supposed to accomplish, let's give ASM (or MASM, if you have chosen to use it) a try and see how it works in real life. Our first program will be the very simple one we started off with in chapter 2: putting the happy face on the screen.

Organizing Your Disk

In writing this book we used a PC with double-density, double-sided disk drives. There is plenty of room on these diskettes for all the files we'll be using to assemble programs. These files are:

ASM

Word Processor Program (including overlays)

LINK

EXE2BIN

DEBUG

The ASM file for our program

The OBJ file for our program

The EXE file for our program

The COM file for our program

The LST file for our program (more on this later)

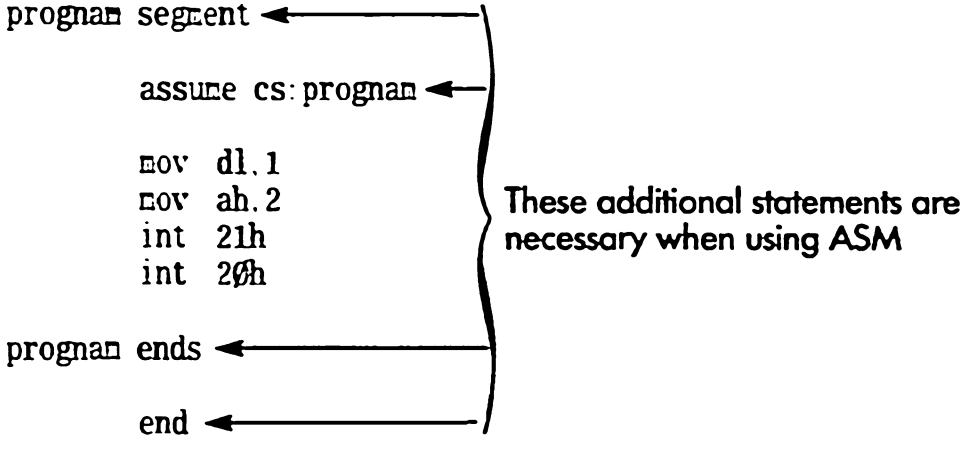
However, if you have disk drives with smaller capacity, you may need to distribute these programs between two different diskettes. You could, for instance, keep the word processing program and the ASM files of your programs on one drive, and the ASM, LINK, and EXE2BIN and DEBUG programs on the second drive.

There are too many variables here to recommend a system that will apply to everyone. In this book we'll assume that all programs are on a single diskette in the A drive.

Creating the ASM File

Set up your word processing program (or EDLIN) to create a file called HAPPY2.ASM. (The "2" distinguishes this program from the one you wrote earlier.) Note that you don't have a choice about the file extension ASM: all files that will be used as input to the assembler *must* have this extension.

Now type in the following program:



(Note that the segment name used in the first, second, and next-to-last lines of the program is "progname", not "program".)

A tab setting of 8 on your word processor is useful in spacing over to the column where the instructions are, although as far as the assembler is concerned a tab or any number of spaces are equivalent. We use two spaces between the *operator* (the instruction itself, like "MOV" or "INT") and the *operand* (the thing the operator operates *on*, like "dl,1" and "21h"), but here again the assembler will like one or more spaces or tabs just as well.

The blank lines are inserted for clarity: to separate parts of the program that do different things. You can put them in with your word

processor by simply hitting ↵ twice at the end of a line.

Once this program is typed in as shown, and saved to your disk as HAPPY2.ASM, it becomes the *source file* for your program. We'll explain how to assemble it in a moment, but first let's make sure we understand all of the details.

You should recognize the entire middle part of the program: it's almost exactly what you typed into DEBUG using "A" in order to create the happy face program in chapter 2.

Using Numbers in ASM

There is one subtle difference, though. Can you find it? That's right: the numbers 21 and 20 are followed by an "h". This is because *ASM assumes numbers are decimal unless told otherwise;* whereas DEBUG always uses hex numbers. This doesn't matter on smaller numbers, like the 1 and the 2, which are the same in decimal and hex, but it's very important for all numbers over 9. It's also very easy to forget, so if your program doesn't work, this is one of the first things to look for: Are all hex numbers followed by "h"?

> When using the assembler, all hexadecimal numbers must be followed by the letter "h".

Although the middle part of the program is roughly the same as the DEBUG version, four new lines have been added — two at the beginning and two at the end of the program. Let's take a look at them and see what they do.

Defining the Segment

Two of the new program lines form a pair. These are:

```
prognam segment
```

and

```
prognam ends
```

These lines tell ASM and LINK that the program will be located in a segment called "prognam" (for "PROGram NAMe"). This can be any name you like, but these two statements must be here, and the *same* segment name must be used in both. Later, when we talk about memory segmentation, we'll find out what these lines really do. Now however you

must accept them on faith: they're there, they must be there, and that's how it is.

Similarly the line

```
assume cs:prognam
```

must also be included at the start of the program. It tells the assembler that the segment where the program is, "prognam", is the one which will be in the CS (Code Segment) register. It's not clear why the assembler can't make this assumption by itself, but it can't, so again this line must be here, and that's how it is.

The END Pseudo-Op

In the last chapter you learned about the "DB" pseudo-op, which was an instruction to the assembler (either the DEBUG "A" command or to ASM) to put certain bytes into memory. END is also a pseudo-op, or instruction to the assembler. Its purpose is very simple: it tells ASM that the program is over, and not to expect any more program lines. Failure to include END at the end of your program will cause the assembler to generate petulant error messages, although it will still assemble correctly. Later we'll see that END can also be used to specify the starting point of a program.

Now you know about all the parts of the program in the ASM file. Remember the four new lines you've learned:

```
prognam segment
assume  cs:prognam
prognam ends
        end
```

These same lines (with perhaps a different segment name) will appear, always in the same order, in all the programs you write to generate COM files.

Using ASM to Create the OBJ File

The next step is to assemble the program. We'll assume that you have HAPPY2.ASM (the file you have just created), the program ASM (or MASM), the program LINK, and the program EXE2BIN on the disk in drive A.

You then type:

```
A>asm happy2
```

Notice that *you don't type the file extension following the file name.* The assembler *assumes* that any file it's going to assemble has the extension ASM. If you type in the extension, the assembler will be confused and trouble can result.

> Don't type the file extension of the file you are about to assemble with ASM (or MASM).

The assembler will respond with a sign-on message, and then ask you to tell it the names of three files: the OBJ file to be generated (as we discussed above), the *listing file* (LST), which we'll ignore for the moment but come back to later, and the *cross reference* file (CRF), which we won't worry about for the moment either. The assembler has already decided what names you are probably going to give to these files, and included them in square brackets as *default* values. ("Default" means what you get if you don't specify something else.) These default values are HAPPY2.OBJ, NUL.LST, and NUL.CRF. The name "NUL" means "no file," so in fact no list or cross-reference files will be generated if these NUL names are used. To invoke these default values, all you have to do is press ↵ after each of the colons.

Here's what the assembler prints:

```
The IBM Personal Computer Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Object filename [HAPPY2.OBJ]:      <— Press ENTER for HAPPY2.OBJ
Source listing  [NUL.LST]:         <— Press ENTER for no LST file
Cross reference [NUL.CRF]:         <— Press ENTER for no CRF file

Warning Severe
Errors  Errors
0       0
```

Once you've pressed ↵ three times, ASM will go on to assemble the program. This will take a few seconds, and you'll hear the disk clicking as it reads the ASM file and writes the OBJ file.

ASM Error Messages

Once the program is assembled the number of "Warning Errors" and "Severe Errors" will be printed out. This is the moment of truth, and it's hard not to feel a little adrenaline rush while you wait to see whether your assembly will be perfect, or whether there are all sorts of disastrous

errors that will involve lots of puzzlement and extensive rewriting of your program. A silent prayer is often helpful here, as the assembler goes about its work.

Warning errors are caused by things the assembler doesn't understand, but which it doesn't think are serious enough to keep the program from assembling correctly. *Severe errors* are those which it thinks will result in a nonfunctional program. You should go back and fix up either kind of error before going on to use LINK.

If there are errors, their reference numbers (or a brief description, if you're using MASM) will be printed out immediately following the program line in which they occur. You can cause these errors to be printed to your printer if you toggle it on with Ctrl PrtSc before you call ASM. (They will also be included in the LST file, which will be generated if you type HAPPY2.LST in answer to the second file name. We'll get to the LST file soon.) All the errors and a few lines describing likely causes for them are listed in appendix A of the *IBM Personal Computer MACRO Assembler* manual. You may spend many happy hours browsing through that appendix, so you should make sure you can find it.

At this point any errors should be the result of mistakes made while typing in the program. A popular error is number 10, "Syntax Error," which occurs when the assembler can't figure out what you're trying to say. However, typing mistakes can generate all sorts of strange messages as well, some of which may be very obscure, so the first thing to do if errors occur is to proofread your ASM source file. When you find the error, correct it with the word processor, then go back and reassemble the ASM file. Do this until you achieve an error-free assembly.

The result of a successful assembly is the HAPPY2.OBJ file. When the assembly is completed, you can use DIR to look for this file on your diskette.

Using LINK to Create the EXE File

Assuming that you now have the HAPPY2.OBJ file on your disk, the next step is to convert the OBJ file to an EXE file, using the LINK program.

Type the following:

```
A>link happy2
```

Again, you *must not use a file extension.* LINK is expecting an OBJ file, and telling it that it has one will only confuse it. The linker will respond with messages similar to those of the assembler. The "run file" is the

EXE file that we want. The list file is useful if we are linking larger programs together, but will not be useful now. Libraries are groups of programs that you may call on in special circumstances to combine with your program (such as those used when linking Pascal programs) but will not concern us here. So, as you did with ASM, you simply press ↵ in response to all three file names:

```
IBM Personal Computer Linker
Version 2.ØØ (C)Copyright IBM Corp 1981, 1982, 1983

Run File [HAPPY2.EXE]:          <— Press ↵ for HAPPY2.EXE
List File [NUL.MAP]:            <— Press ↵ for no MAP file
Libraries [.LIB]:               <— Press ↵ for no LIB files
Warning: No STACK segment

There was 1 error detected.
```

The phrase "Warning: no STACK segment," and the fact that there was "1 error detected" don't really mean anything to us at this point. Stack segments are not used in COM files, so there's no way we can get rid of these "error messages." This is unfortunate, since they sit there making us feel a little nervous and guilty, even after we've proved that our program is going to run just fine.

The linker can generate all sorts of other error messages as well, but if you've typed in the above program correctly they should not appear.

The result of this LINKing process should be the file HAPPY2.EXE on your disk. Make sure it's there with DIR.

Using EXE2BIN to Create a COM File

EXE2BIN is easier to use than ASM or LINK. All you do is type "exe2bin", then the name of the EXE file you want to convert, again *without the file extension*, and finally the name of the file you want to convert it to, this time (just to keep you on your toes) *with the file extension.*

```
A>exe2bin happy2 happy2.com
          ↑      ↑
      EXE file  COM file
  (no extension) (extension.COM)
```

The default value of the second name is a file with an extension of BIN, so if you leave the second name out, (like this):

```
A>exe2bin happy2
```

you'll get a HAPPY2.BIN file. If you then want to run this directly from DOS you'll have to convert it to a COM file with the RENAME utility. So the first approach — typing in the desired COM file — is simpler.

Running the Program

You should now have a file called HAPPY2.COM on your diskette. You can execute this file directly from DOS, simply by typing the filename — no extension is required.

```
A>happy2        <— You type the program name
☺               <— Happy face appears
A>              <— Returns to DOS
```

You can also execute the program from DEBUG, after loading it in when you call up DEBUG. In this case, you *do* need the file extension. (Learning when to use extensions and when not to use them is, as you can see, part of an elaborate initiation ritual.)

```
A>debug happy.com                <— You type this to load program
-g                               <— You type this to execute program
☺                                <— Program prints happy face
Program terminated normally.     <— Return to DEBUG
-                                <— New DEBUG prompt
```

What Good Is It, Anyway?

So, after all that, we're just about back to where we were when we typed the program in using the "A" option in DEBUG. It's a lot more complex using this assembler approach to go from typing in the program to executing it, and there don't seem to be any benefits. So far, that's true. The advantages of the assembler only start to be apparent when we get to longer programs. However, before we plunge into really big programs, we need to develop our familiarity with the assembler and the assembling process.

Let's do a slightly longer program now — one which shows off (among other things) one of the really nice features of the assembler: symbolic addressing, which is using *names* for addresses instead of numbers.

Assembling SMASCII2

Remember SMASCII, the program that printed out all the ASCII characters just once and then returned very nicely to DOS or DEBUG? Hop into your trusty word processor and type it in (using the assembler format from the last example) like this:

```
prognam segment         ;start of segment

        assume cs:prognam  ;assume what's in CS

        mov  cx,1ØØh    ;put count in CX
        mov  dl,Ø       ;first ASCII character
next:
        mov  ah,2       ;Display Output funct
        int  21h        ;call DOS to print
        inc  dl         ;next ASCII character
        loop next       ;do again, unless done

        int  2Øh        ;return to DOS

prognam ends            ;end of segment

        end             ;end of assembly
```

Comment Fields

As you can see, there are several new things in this program. The most obvious is that the program lines have comments added to them. This is a really nice aspect of assembly language, as compared at least with interpreted languages like BASIC. You can put as many comments (each preceded by a semicolon ";") in the source file as you like — they won't add anything to the OBJ, EXE or COM files. All they do is make the program much easier to understand.

Not all programmers add a comment to every line, as we have done here, but they probably should. An assembly-language source file is not necessarily an easy thing to understand, and the more comments you add the easier it becomes. Don't forget that — even if no one is ever going to look at your program but you — what's clear to you now may be obscure when you go back to make a change in the program. And if anyone else is ever going to look at your listings, comments are even more important.

> You can never have too many comments in an assembly-language source file.

As an example of good use of comments, look at the ROM BIOS listing in appendix A of the *IBM Personal Computer Technical Reference* manual. Every line (unless it's part of a table of similar items) has a comment.

If you think this sounds like we're exhorting you to use a lot of comments in your listings, you're right. No one was ever sorry for having explained too many things, but the wailing and lamentation of programmers who didn't use enough comments can be heard throughout the land.

The rule for using comments is very simple: *comments must start with a semicolon.* They can occur anywhere on a line, but everything following the semicolon will be treated as a comment; that is, it will be ignored by the assembler. The usual places to start comments are at the beginning of the line, or at the beginning of the *comment field.*

What's a "field"? It's simply one of the parts of a line of assembly code. There are four fields:

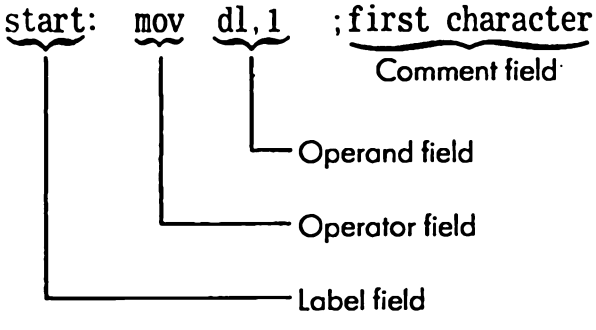
The label field

The operator field

The operand field

The comment field

Here's an example of an instruction that uses all four fields:



You've met operators and operands before. The comment field is to the right of the operand field, and is separated by one or more spaces or tabs. What about the label field? Read on.

Symbolic Addresses

You've probably noticed another new aspect of the program: the name "next:" halfway through the program, and the "loop next" at the bottom. What's that all about?

In the DEBUG version of this program we said "loop 105", in order to jump back and display the next character. You can see by comparing the two versions that the word "next" now *stands for* the address 105. The advantage of using a symbolic name is that when we're writing the program we don't need to know, or care, what actual address we'll be jumping back to. We just put the *label* — as the "next:" in the left column is called — at the appropriate line in the program. Then whenever we want to refer to this place in the program, we don't need to know its address, we simply refer to the name used as the label.

One of the more obvious benefits of labels is that it's easier to change our program — by adding or deleting a line, for instance. Suppose we added an instruction to the early part of the SMASCII program so that instead of LOOPing back to 105, we were LOOPing back to 107. If we were using *numbers* for addresses we would have to change the LOOP instruction itself, to LOOP 107. But if we're using labels no change is necessary, since the *assembler* will now figure out where the "NEXT:" label is, and modify the LOOP NEXT instruction accordingly.

The label can be quite long — up to 31 characters. (Actually it can be longer, but only the first 31 characters are recognized by the assembler.) This makes it possible to use very descriptive labels, like "line_input_flag". (The words of multi-word labels can be separated by underline symbols.) However, the use of very long names tends to make it harder to produce an organized looking listing, since it's harder to get the columns to line up. In this book we'll stick to fairly short names, but for an example of some longer names, look at the ROM BIOS listing in your *IBM Personal Computer Technical Reference* manual, and at the *Bluebook of Assembly Language Routines for the IBM PC and XT* by Christopher L. Morgan (New York: Plume/Waite, New American Library, 1984).

The label field is the first one on each line. It's separated from the operator field by one or more spaces or tabs, as usual. We'll show you many more examples of labels used for symbolic addresses as we go along. A label field can either go on the same line as the instruction it labels, or it can go on the preceding line.

"Symbolic Addressing" means using a name to refer to an address rather than the actual hexadecimal number.

We're going to assemble SMASCII2 in just the same way as we did HAPPY2, with one exception: we're going to generate a new file, one of the ones for which we used the default value of NUL last time. This is the LST, (for LiST), file. The reason we want to produce this file will be obvious once we see it.

So, with your SMASCII2.ASM file and ASM and LINK on the same disk, call up ASM and answer its questions the same way as before, except that for the second file name, instead of hitting ↵, type the program name. You can also add the file extension LST, but it's not necessary, since if you don't type it, "LST" will be added automatically.

The LST File

Now if you use DIR to look at the directory you'll see a new file: SMASCII.LST. Use the TYPE function from DOS to type it out. It should look like this:

```
                                                    PAGE    1-1

0000                        prognam segment         ;start of segment

                                    assume cs:prognam ;prognam in CS

0000  B9 0100                       mov  cx,100h    ;put count in CX
0003  B2 00                         mov  dl,0       ;first ASCII character
0005                        next:
0005  B4 02                         mov  ah,2       ;Display Output funct
0007  CD 21                         int  21h        ;call DOS to print
0009  FE C2                         inc  dl         ;next ASCII character
000B  E2 F8                         loop next       ;do again, unless done

000D  CD 20                         int  20h        ;return to DOS

000F                        prognam ends            ;end of segment

                                    end             ;end of assembly


                                                    Page    Symbols-1
```

```
Segments and groups:

                N a m e                 Size    align   combine class

PROGNAM. . . . . . . . . . . . .        ØØØF    PARA    NONE

Symbols:

                N a m e                 Type    Value   Attr

NEXT . . . . . . . . . . . . . .        L NEAR  ØØØ5    PROGNAM

Warning Severe
Errors  Errors
Ø       Ø
```

Well, isn't that the slickest thing you ever saw? All the addresses and actual machine language are printed out next to the symbolic instructions that represent them. This is a fascinating listing, because it shows us what the assembler did. The input to the assembler, the source or ASM file, is represented by the columns on the right, and the output — the OBJ file — is represented by the columns on the left. There is a lot of space between these two columns because the assembler, as you will see later on, often generates a lot of hex numbers for a single assembly-language instruction.

This part of the LST file, showing the program, is the bread and butter file when debugging (finding the errors in) assembly-language programs. To debug a program, or even to understand it completely, it's usually necessary to be able to look at both the assembly-language and the machine-language versions of the program side by side. The machine language by itself is too obscure and hard to understand, and the assembly-language object code doesn't tell you what the assembler program has really placed in your computer's memory. You will generally need to print out the LST file on your printer to get the full benefit from it, although it is possible to debug programs without the printed version. In the next chapter we're going to introduce DEBUG's powerful "T" (for "trace") command, and then this LST printout will prove invaluable for following what is really going on in your program.

Following the program are two lists called "segments and groups" and "symbols." The segment name you used in the program is the only entry in the first list, and the label "next" is the only entry in the second. These lists are generated because in larger programs it often helps to be able to quickly look up the location of a particular symbol or segment name. For

the smaller programs we will be concerned with, these lists are not so important, and can usually be ignored.

Linking SMASCII2

The linker is used just as it was before:

```
A>link smascii2

IBM Personal Computer Linker
Version 2.ØØ (C)Copyright IBM Corp 1981, 1982, 1983

Run File [SMASCII2.EXE]:      <— Press ENTER
List File [NUL.MAP]:          <— Press ENTER
Libraries [.LIB]:             <— Press ENTER
Warning: No STACK segment

There was 1 error detected.
```

And again this should generate an EXE file, whose existence you can check using DIR.

Converting SMASCII2 with EXE2BIN

To convert the EXE file to a COM file, type:

```
A>exe2bin smascii2 smascii2.com
```

Executing SMASCII2

As before, SMASCII2 can be executed either from DOS or from DEBUG. From DOS, type:

```
A>smascii2
```

and from DEBUG, type:

```
A>debug smascii2.com

-g
```

Either way the program should run just as it did before, when you created it with DEBUG's "A" command. If not, back to the old drawing board. There's probably a typo somewhere.

Deciphering Machine-Language Op-Codes

Now that you have a LST file to study, you may have begun to wonder exactly what the correspondence is between the assembly-language mnemonics and the hex numbers in the resulting machine code. For instance, you have probably noticed that INT 20 always translates into the hex number CD followed by the hex number 20. CD is obviously the op-code for INT, and the 20 is simply an 8-bit number, so this instruction isn't hard to figure out.

However, for other instructions, like MOV, things can get rather complicated, and all sorts of different hex numbers can turn up in the machine language translation. In the SMASCII listing above, for example, B9, B2, and B4 are all used for the MOV instruction. How does the assembler know which of these numbers to use? And do we really need to know how it knows this?

We aren't usually going to be too concerned with the actual numbers stored in our computer's memory. After all, you have the assembler to generate these numbers for you, and if you have some numbers and you want to know what they mean, you can always use the "U" command to disassemble the code. For this reason we don't show the machine language equivalents in the instruction summaries in this book.

A Simple Example

Once in a while, however, it's useful to know exactly what the binary equivalent of a certain instruction is supposed to be. Figuring it out can be an obscure process, but let's run through it for an example instruction, say the MOV CX,100h instruction which is the first one in the SMASCII program above. The machine-code equivalent of this instruction, as shown in the LST file, is B9 0100. The 0100 is clear enough — it's simply the constant that will be MOVed to CX. But where does the B9 come from?

First, look up MOV in the *IBM Personal Computer MACRO Assembler* manual. You'll find several sections devoted to this instruction. The one you want is "TO Register FROM Immediate-data". Under "Encoding" you'll see the cryptic notation:

1011wreg data data-high*

*present only if w = 1

Let's analyze this. The first four digits, 1011, are binary for the hex number "B". That explains where the first digit of the B9 0100 comes from. The "w" means "word". The 8088 needs to know whether it's

supposed to be MOVing a byte or a word, and "w" represents a single bit which tells it. If this bit is on (set to 1), the data is a word. If it's off (cleared to 0), then the data is a byte.

How about the "reg"? As you may have guessed, this stands for "register". If you look at the beginning of the chapter on instruction mnemonics in your *IBM Personal Computer MACRO Assembler* manual, you'll see a table showing the "reg" values for all the registers and their numbers. There are actually two tables, one for 8-bit registers (like CL), and one for 16-bits (like BX). Our instruction is going to put a word into the 16-bit register CX, so what we want is the 16-bit code for CX, which is given as the 3 digit binary number 001.

Putting all this together, we can now construct the first two digits of the op-code, as shown in Figure 5-4.

So that's how the hex number B9 0100 is constructed. Unfortunately, there are even more complicated examples.

A More Complicated Example

When indirect addressing is used in an instruction the calculation of the machine language op-code becomes somewhat more involved. Take for example the MOV DL,[BX] instruction at address 0106 of the PRINTHI2.COM program. (As you recall, this instruction uses indirect addressing, signaled by the square brackets, to MOVe the contents of the memory location contained in BX, and place it in the DL register.) The machine language equivalent of this instruction, as shown in the "U" listing for the PRINTHI2.COM program, is 8A17. Where do these numbers come from?

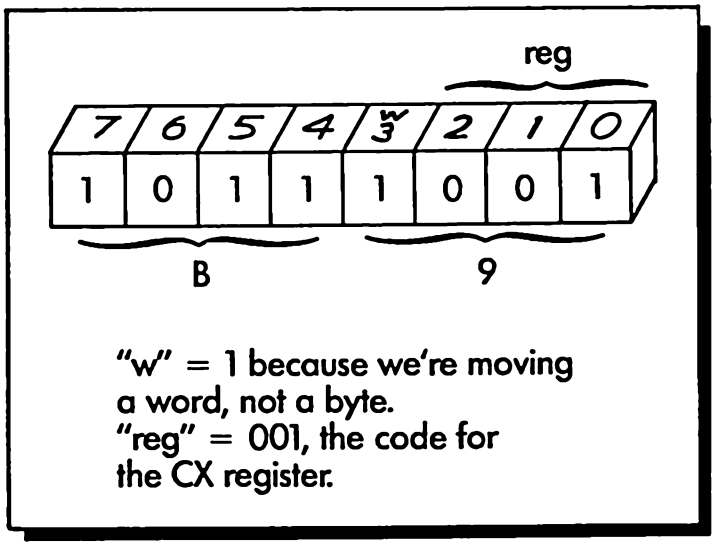


Figure 5-4. Machine code of the MOV CX,0100 instruction

Again, look up the MOV instruction in the *IBM Personal Computer MACRO Assembler* manual. This time we want the section called:

TO Register FROM Register

TO Register FROM Memory-or-Register Operand

TO Memory-or-Register Operand FROM Register

The encoding here is given as:

100010d1 modregr/m addr-low* addr-high*

*these bytes omitted in register to register moves, when...

Now if that doesn't look like an expression from genetic engineering, what does? Let's see if we can make sense of it.

Figure 5-5 shows what is meant by the various notations in the two-byte form of this instruction. The addr-low and addr-high are not used in the particular case of MOV DL, [BX], because we are not using what is called the "displacement," which is simply an address like 0200 (or a label representing the address).

As you can see, the first six bits of the instruction are always the same. The next bit is the "d" bit, which stands for "Direction." If d = 1, data goes from memory to a register; if d = 0, data goes from a register to memory. The term EA is used to refer to memory. It stands for "Effective Address," which means the actual memory address where the data is to be placed (or taken from). This EA may be found by combining several different things, like a displacement, and the number in BX, and

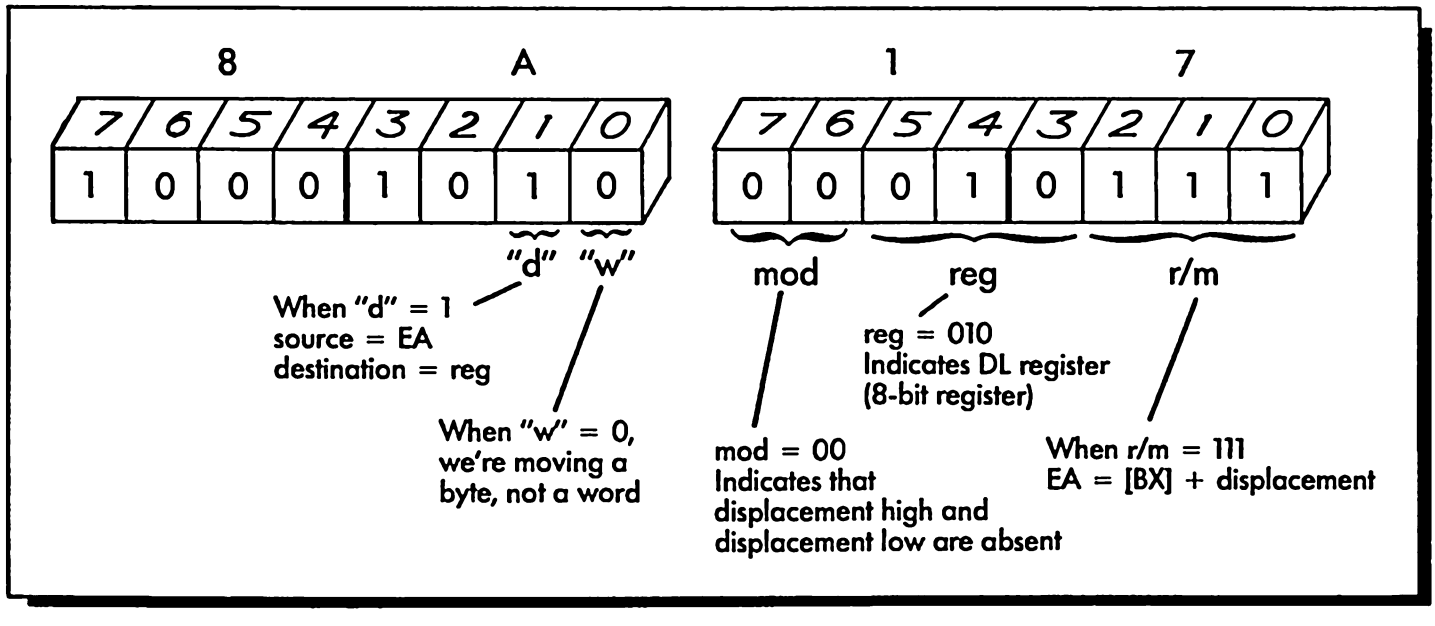


Figure 5-5. Machine code of the MOV DL, [BX] instruction

the number in another register like SI. (We'll see examples of these more complex addressing modes later in the book.) In our case the EA is simply the number in the BX register.

Next we have the "w" bit, which we have discussed. It is 0, because we are dealing with a byte. (In our edition of the *IBM Personal Computer Macro Assembler* manual there is a misprint for the value of this bit: it is shown as a "1", instead of a "w" as it should be.)

Now we get to a new designation, the "mod" field. "Mod" stands for "MODify," and is used to indicate whether displacement fields are part of the instruction, and if so whether there will be high and low displacements, or just low. We are not using displacements, so this field is set to 00. The various values of the "mod" field are listed at the beginning of the chapter on instruction mnemonics in the *IBM Personal Computer MACRO Assembler* manual.

We've talked about the "reg" field. In our example the code 010 indicates the DL register, since we are talking about 8-bit registers (as specified in the "w" bit).

Finally we have a new field, "r/m," which stands for "register/memory." The various values this field can take on are also listed in the instruction mnemonics chapter referenced above. In our case r/m = 111, which means EA = [BX] + displacement. Since we are not using a displacement (as indicated by the "mod" field), this can be interpreted as EA = [BX].

Putting all these fields together, as shown in Figure 5-5, we arrive at the hex number 8A17 for the machine-language translation of MOV DL,[BX].

Whew! Imagine how hard the MACRO Assembler has to work to figure this kind of thing out, dozens of times a second. Remember, you will very seldom need to go through this process — but now you know how to do it.

For more detail on this topic see *8088/8086 16-bit Microprocessor Primer* by Christopher L. Morgan and Mitchell Waite (New York: Byte/McGraw-Hill, 1982).

Using a Batch File to Speed Assembly

We've assembled two programs in this chapter, and the steps have been the same for both. We call up the assembler with our program name, and then hit ↵ for the three questions the assembler asks. Then we do the same thing with the LINK program, and finally we call up EXE2BIN with the name of our program followed by the name of the

resulting COM file. All this typing can get pretty repetitious. Is there a way to automate it? After all, we are using a computer!

The *batch processing* feature of DOS is exactly what's called for here. (If you're unfamiliar with this DOS capability you should read about it in *IBM's Personal Computer Disk Operating System* manual.) To use batch processing, you first create a file which contains a number of commands *to DOS*. When this file is subsequently executed, the commands will be carried out, one after the other, quickly and without typing errors, while you sit back and enjoy a cup of coffee.

The Command Line in ASM

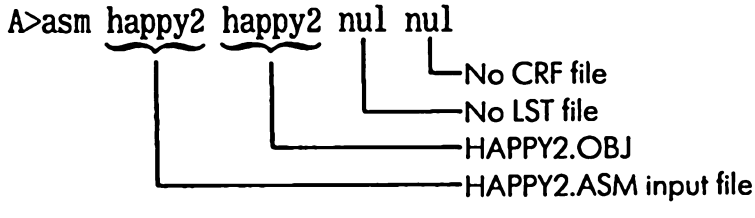
Before you can create the appropriate *batch file*, you need to be aware that ASM (or MASM) can be operated in two different modes. In the mode you've been using so far, you simply type ASM followed by the name of the program. Then when the program starts to execute, it asks you questions about the names of the files you want generated, as we saw earlier:

```
A>asm happy2

The IBM Personal Computer Assembler
Version 1.ØØ (C)Copyright IBM Corp 1981

Object filename [HAPPY2.OBJ]:          ← Press ↵ (or enter HAPPY2)
Source listing  [NUL.LST]:HAPPY2       ← Press ↵
Cross reference [NUL.CRF]:             ← Press ↵
```

But there's another way to tell ASM what output files we want. We can enter all the information on the command line, immediately following the name of the program we want to assemble:



This second mode comes in handy, since a command line such as that above is *a command to DOS*, and can therefore be executed in a batch file.

The Command Line in LINK

The LINK program also required us to press ↵ three times once it was loaded:

```
A>link smascii

IBM Personal Computer Linker
Version 2.ØØ (C)Copyright IBM Corp 1981, 1982, 1983

Run File [SMASCII2.EXE]:      ←— Press ↵
List File [NUL.MAP]:          ←— Press ↵
Libraries [.LIB]:             ←— Press ↵
Warning: No STACK segment

There was 1 error detected.
```

LINK requires a somewhat more complex approach to be used in "command line" mode than ASM did. Instead of putting the names of the output programs on the command line as we do with ASM, it's necessary to create a separate file called AUTOLINK, which has *no file extension*. We need to put the names of the output files in AUTOLINK, or blank lines (press ↵) if we want their default values. For different uses of LINK this AUTOLINK file can be very complex, but for our purposes all we want to do is tell LINK to use the four default file names; so we want to send it four carriage returns (↵ s). We thus need a file called AUTOLINK with four carriage returns in it.

EDLIN is perfect for creating such a file. Once created, the file looks like this when we examine it again with EDLIN:

```
A>edlin autolink
End of input file
*1
     1:*
     2:
     3:        ←— Four carriage returns ( ↵ s)
     4:
*
```

Creating Batch Files

Batch files use the notation "%1", "%2", and so on to refer to the file names typed in by the user when the batch file is executed. Let's take a look at a batch file we can use to turn an ASM file into an COM file, to get an idea how it all works.

Here's the batch file:

```
asm %1 %1 nul nul       <— Create OBJ file with same name as ASM file
link %1 @autolink       <— Create EXE file with same name as OBJ file
erase %1.bak            <— Erase BAK file produced by word processor
erase %1.obj            <— Erase OBJ file
exe2bin %1 %1.com       <— Convert EXE file to COM file
erase %1.exe            <— Erase EXE file
```

We create this file using EDLIN or a word processor, and we give it the name COMASM.BAT. (You can make up your own name if you prefer, but you must use the extension BAT.) In order to use this file all we have to do is enter COMASM followed by the name of the ASM file which we want to assemble automatically, link and convert into a COM file. For instance, to use this batch file on the SMASCII program, we would enter

```
A>comasm smascii
```

That's all we have to do! The COMASM batch file will take care of the entire process for us.

The first line of the batch file causes ASM to generate a SMASCII.OBJ file. The second line causes LINK to generate an EXE file. Now, our particular word-processor program generates two files when we create a text file: the second is a backup file with the extension BAK. We want to get rid of this file to save space on the disk, so the next line erases SMASCII.BAK. We've already used the OBJ file, SMASCII.OBJ, so we erase that too, in the next line. The next to last line of the program tells EXE2BIN to convert SMASCII.EXE to SMASCII.COM, and finally we erase SMASCII.EXE.

So, when you start this process you'll have a single file, SMASCII.ASM. At the end of the process you'll have two files: SMASCII.ASM and SMASCII.COM. All the intermediate files have been erased to save space on the disk.

We recommend you create your own batch file and use it in the example programs that follow in this book. The time it saves will amply reward your efforts in creating it.

Summary

In this chapter you've learned how to write programs in a form suitable for assembly by the IBM MACRO Assembler, and how to assemble the programs. You've also learned how to change the OBJ file to an EXE file with LINK, and how to change the EXE file to a COM file with EXE2BIN. You've been introduced to the idea of symbolic

addressing, and you know how to create a LST file. You've learned how the MACRO Assembler figures out the machine-language numbers that correspond to the symbolic instructions in the source code, and finally you've learned how to use batch processing to simplify the assembly process. In the next chapter you're going to put what you've learned here to use.

6

Using the IBM MACRO Assembler

Concepts
- Two's complement arithmetic
- Flags
- Conditional jumps
- Subroutine calls
- Cross-reference file

Debug Commands
- T = Trace
- Breakpoints using G

8088 Instructions
- ADD = Add
- ROL = Rotate left
- DEC = Decrement
- JNZ = Jump if not zero (or JNE = Jump if not equal)
- CMP = Compare
- JL = Jump if less than (or JNGE = Jump if not greater nor equal)
- NOP = No operation
- SUB = Subtract
- JG = Jump if greater than (or JNLE = Jump if not less nor equal)
- CBW = Convert byte to word
- XCHG = Exchange registers
- MUL = Multiply
- PROC = Define a procedure (pseudo-op)
- CALL = Call a procedure
- RET = Return from a procedure
- PAGE = Number program lines (pseudo-op)

Applications
- BINIHEX routine — Binary to hexadecimal converter
- DECIBIN routine — Decimal to binary converter
- DECIHEX program — Decimal to hexadecimal converter

In the last chapter you learned the fundamentals of using the IBM MACRO Assembler (either ASM or MASM, depending on the size of your system and your own preference). In this chapter we're going to put what you learned to work.

We'll start off by writing a short program called BINIHEX, which will take a binary number stored in the BX register, and print it out on the screen as a hexadecimal number. This program will introduce some new 8088 instructions, and will also introduce you to the use of the "T" (for "Trace") command in DEBUG, which is a very powerful debugging tool.

Next we'll write a second program, DECIBIN, which takes a decimal number you type in at the keyboard, and converts it to a binary value in the BX register. Again, this program will introduce new 8088 instructions.

Finally (did you guess this?) we'll put BINIHEX and DECIBIN together in a larger program, called DECIHEX, which will take the decimal number you type at the keyboard, and print the hexidecimal equivalent out on the screen. You will thus end up with a useful utility program which can be called directly from DOS: a decimal to hex converter, for those of you who are tired of looking things up in tables.

The BINIHEX Program

The purpose of BINIHEX is to take a binary number from a register in the 8088, and print it out on the screen, in hex. As an example of the usefulness of this kind of routine, look at the DEBUG program: it uses a similar routine to print out the addresses, and the contents of the addresses, whenever you use "D" (and most of the other commands as well). BINIHEX is a sort of window into the 8088: with it, we can examine the bits that constitute the contents of the otherwise invisible registers in the microprocessor chip, and display them on the screen. We'll find a variety of uses for this routine throughout the book, including the decimal to hex conversion routine.

Writing the Source File

Instead of showing you the ASM file as you should type it in, we're going to do things a little differently and show you the LST file which is produced when you assemble the program. As you recall, this file has the machine language on the left and the original assembly language on the right. Using the LST file has the advantage of letting you see right away what the machine language for the program looks like, and saves us

showing two listings, one for ASM and one for LST, which contain much of the same information. We'll follow this procedure from now on in this book. Here's the listing:

```
0000                            prognam segment         ;start of segment

                                        assume cs:prognam

0000  B5 04                             mov  ch,4       ;number of digits
0002  B1 04                     rotate: mov  cl,4       ;set count to 4 bits
0004  D3 C3                             rol  bx,cl      ;left digit to right
0006  8A C3                             mov  al,bl      ;move to AL
0008  24 0F                             and  al,0fh     ;mask off left digit
000A  04 30                             add  al,30h     ;convert hex to ASCII
000C  3C 3A                             cmp  al,3ah     ;is it > 9 ?
000E  7C 02                             jl   printit    ;jump if digit =0 to 9
0010  04 07                             add  al,7h      ;digit is A to F
0012                            printit:
0012  8A D0                             mov  dl,al      ;put ASCII char in DL
0014  B4 02                             mov  ah,2       ;Display Output funct
0016  CD 21                             int  21h        ;call DOS
0018  FE CD                             dec  ch         ;done 4 digits?
001A  75 E6                             jnz  rotate     ;not yet

001C  CD 20                             int  20h        ;return to DEBUG

001E                            prognam ends            ;end of segment

                                        end             ;end of assembly
```

Of course, to create the ASM file, you type in only the symbolic instructions from the right-hand columns. The numbers in the left-hand columns don't really exist until you've created the ASM file and assembled it.

Type in the program, and then assemble it with ASM, taking care to answer BINIHEX to the question about the LST file name:

```
A>asm binihex                   <-- You enter this

The IBM Personal Computer Assembler
Version 1.00 (C)Copyright IBM Corp 1981

Object filename [BINIHEX.OBJ]:          <-- Press ENTER
Source listing  [NUL.LST]:binihex       <-- Enter the program name
Cross reference [NUL.CRF]:              <-- Press ENTER
```

```
Warning Severe
Errors  Errors
Ø       Ø
```

Now you can print out the LST file, and see if it's just the same as the one shown above.

Once the program is assembled, use LINK to create the EXE file, and then EXE2BIN to create the COM file, just as you did in the last chapter. When you've finished, you'll have a COM file that can be executed from DOS and from DEBUG.

Operating the BINIHEX Program

Before we go on to explain how this program works, we'll tell you how to operate it. The chances are you're going to try running it anyway, right? It's well known that no one ever reads the documentation before putting the product into use.

First, try running the program directly from DOS. You should get a four-digit number printed out on your screen, like this:

```
A>binihex
ØØØØ
```

The only trouble is, this number is always zero! So what good is that? No good at all, if the program is executed by itself, from DOS. However, if we get into DEBUG and start to fool around, we can begin to see what BINIHEX can do.

```
A>debug binihex.com
-g
ØØØØ
Program terminated normally
-
```

Same result. The trouble is, the program is printing out the contents of the BX register, as it's designed to do, and, unless we put something in it, the content of this register is zero. So let's put something in it. Remember that although we'll be typing in a hex number, this number will exist physically in the register in binary. Still in DEBUG, try this:

```
-rbx          ←— You type "rbx" to see the BX register
BX ØØØØ      ←— It's zero
:1234         ←— Change it to 1234h
-
```

Now, run the program again. (You might want to check that the IP register is set to 100 before you do this, as discussed in the last chapter.)

```
-g              <— Type "g" to run the program
1234            <— It prints out the contents of BX
Program terminated normally
-               <— Back to DEBUG
```

So it seems to be working! Try inserting some other numbers, like ffff and abcd, into the BX register, and then running the program. It should print them out too.

It may not seem that this program accomplishes very much. You put a hex number into the BX register with "R," and the program prints it out,

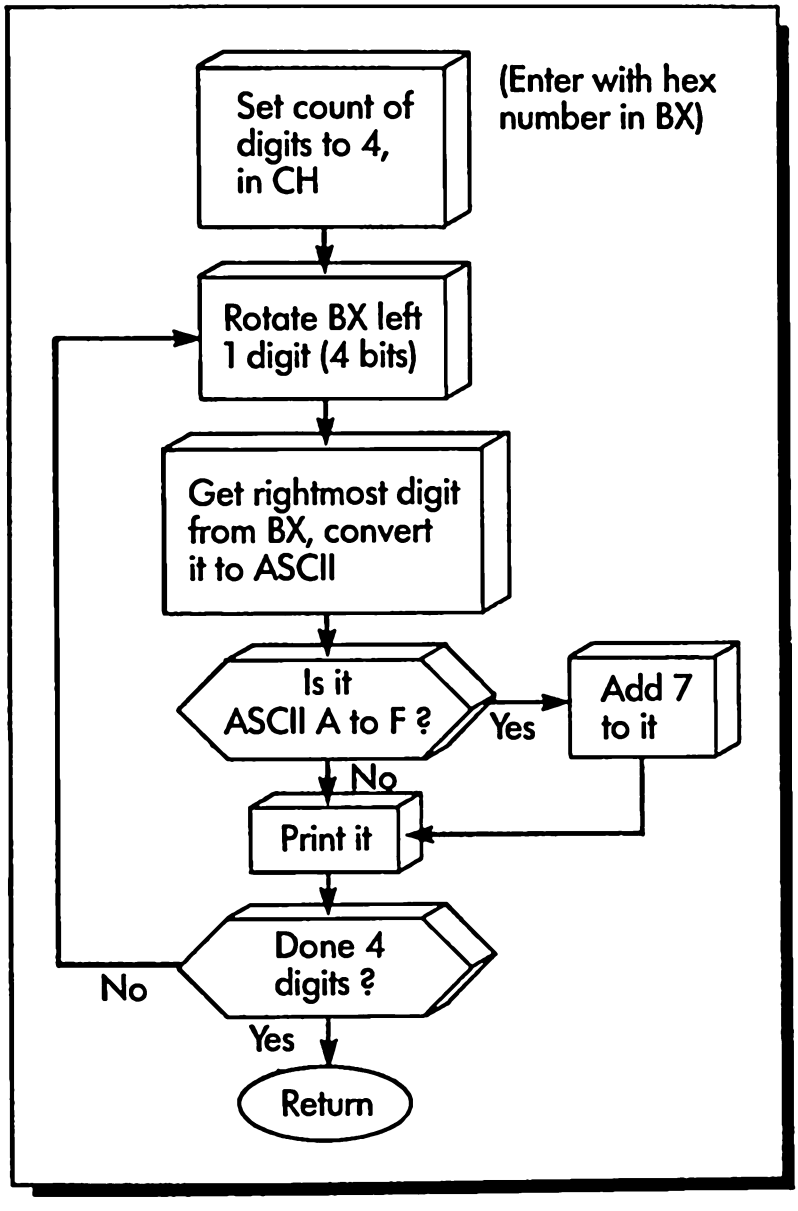


Figure 6-1. Flow chart of the BINIHEX program

unchanged. Later, however, when a binary number has been placed in BX by another routine, printing it out in hex will give us new and important information.

How Does BINIHEX Work?

Let's examine the inner workings of the program. The flow chart of its operation is shown in Figure 6-1.

Rotating the BX Register

We start off with four hex digits (sixteen binary digits) in the register BX. We want to print these four hex digits out, one at a time: first the one on the left, then the second from the left, and so on. The first thing we do is *rotate* the entire contents of BX one hex digit, which is four bits, to the left. That puts the leftmost digit in the right-hand place. For instance, if the number we inserted into BX was 1234, after rotating it one digit to the left BX would contain 2341. (When you rotate the contents of a register, the things that are pushed off one end re-enter at the other end. We'll have more to say about this when we talk about the ROL instruction.) This is shown in Figure 6-2.

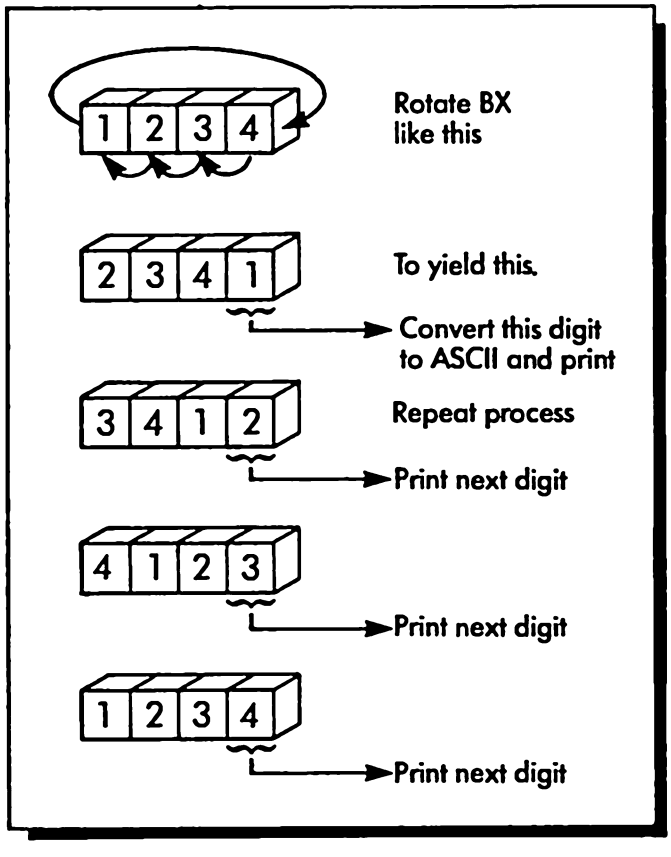


Figure 6-2. Rotating the BX register

Changing the Hex Digit to ASCII

The 1 is the first hex digit we want to print, so we move the BL part of BX into the AL register with a MOV instruction. The AL register is the preferred place to do arithmetic, since this register can generally perform 8-bit arithmetic faster than the other registers, (although they can do most of the things AL can do, in a pinch).

AL now contains 41h. We don't care (yet) about the 4, so we mask off the 4 with the AND instruction using 0Fh (which is 00001111 binary) as the mask.

Now the ASCII value of the printable character "0" is 30h, the ASCII value of 1h is 31h, and so on up to 9h, whose ASCII value is 39h. So to convert these digits to ASCII, we add 30h to them, with the ADD instruction. We are then, except for one small problem, ready to send this number off to the Display Output function to be printed (using the instructions following the label "printit").

The small problem is caused by the fact that we are dealing with hex rather than decimal digits. The ASCII values of the first ten hex digits are in order:

digit	ASCII value (hex)
0	30
1	31
2	32
3	33
4	34
5	35
6	36
7	37
8	38
9	39

And the ASCII values of the last six hex digits are in order:

digit	ASCII value (hex)
A	41
B	42
C	43
D	44
E	45
F	46

But unfortunately there is a gap between the two series of values: the difference between 3Ah (which is one past "9," where "A" ought to be) and 41h (where "A" actually is), is 7. So if the digit we are about to print is a hex digit from A to F, we need to add 7 to it to get the correct ASCII value. To find out if the digit is in the first series (0 to 9) or the second (A to F) we use the CMP (for "CoMPare") instruction, which we'll describe below. If the content of AL is less than 3Ah (the value which is one larger than the ASCII value of "9"), then we need to add the additional 7. Otherwise, we want to jump over this ADD instruction, and go directly on to print the digit.

Do It All Four Times

Once the first digit has been printed out in ASCII, we want to rotate BX again to get the second digit from the left, which is 2, and print that. When we've printed all four digits, we're done.

New Instructions

To perform as described our program needs some new 8088 instructions. Let's look at these new instructions and see how they operate.

The ADD Instruction

The ADD instruction does just what you expect it to: adds the contents of the rightmost operand to the contents of the leftmost operand, and leaves the result in the leftmost operand. Thus, if AX contains 20 and CX contains 30, after you execute the instruction

```
ADD AX,CX
```

then AX will contain 50, while CX is unchanged. Similarly, a constant can be added to a register, as we described with the AND instruction in the last chapter.

For the moment don't worry about the "FLAGS" part of this instruction box. We'll be getting to flags soon.

ADD Instruction

Adds two operands. Result (sum) is stored in leftmost operand.

To add contents of two registers:

```
ADD AL,BL
ADD BX,CX
```

To add constant to register:

```
ADD DL,2Ah
```

To add register to memory:

```
ADD MWORD,DX
```

Also, a number can be added to a memory address, and a memory address can be added to a register.

Flags affected: AF, CF, OF, PF, SF, ZF

Signed Arithmetic

It should be noted that the ADD instruction, like other arithmetic instructions in the 8088, performs *signed* arithmetic. That means that it thinks of the high-order bit (number 7 in 8-bit quantities, and number 15 in 16-bit quantities) as being a *sign* bit. (See Figure 6-3.)

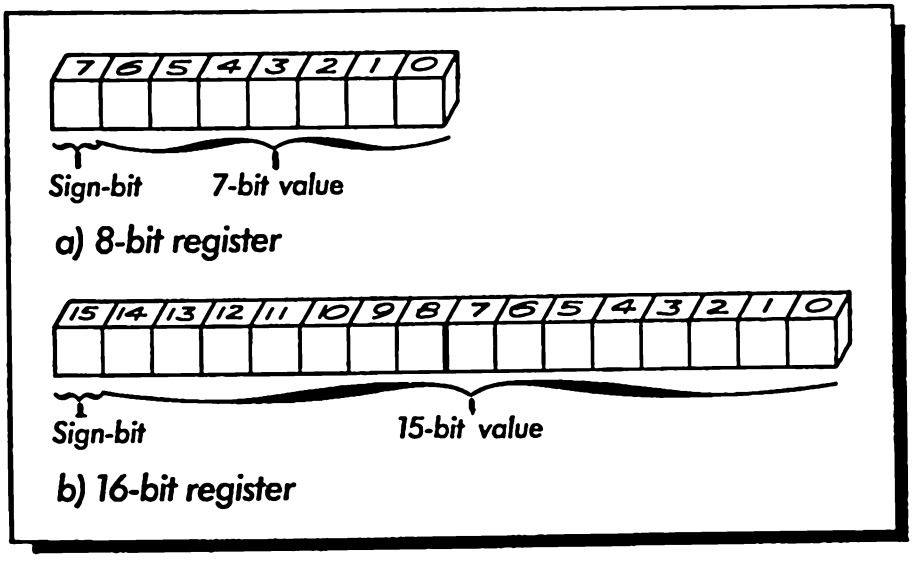


Figure 6-3. Signed numbers

Thus (for 8-bit numbers) the number 01h is just +1, while the number FFh is –1. 7Fh is just 7Fh (127d), while 80h is interpreted as –80h (–128d); 81h is –7Fh (–127d), and so on. This way of representing signed numbers is called "two's complement arithmetic," a phrase which is useful to impress potential employers if nothing else. The table below summarizes the way 8-bit numbers are represented.

Contents of Register	Arithmetic Value	Decimal Equivalent
00	0	0
01	1	1
02	2	2
...	...	...
7E	7E	126
7F	7F	127
80	–80	–128
81	–7F	–127
82	–7E	–126
...	...	...
FD	–3	–3
FE	–2	–2
FF	–1	–1

Similarly, for 16-bit numbers, anything over 7FFFh (32767d) is considered to be negative. If you have the 16-bit number FFF0h, which is –10h, and you add it to 60h with an ADD instruction, the result will be 50h. And if the content of a register is 0000, and you decrement it (subtract 1 from it) with a DEC instruction, it will become FFFFh, which is –1.

You don't really need to be a whiz at this sort of arithmetic to get along in assembly language. The important thing to keep in mind is that the 8088 considers those numbers whose leftmost bit is *set* (= 1) to be negative, and those numbers whose leftmost bit is *cleared* (= 0) to be positive.

The ROL Instruction

The ROL instruction takes the bits in a register and rotates them to the left. "Rotate" means that the bits rotate around the register: they are pushed off one end, and rotate around to the other end. The bit pushed off the end is also placed in the carry flag. (We'll talk about flags in the next section. For the time being, think of the carry flag as a place to store a single bit.)

ROL Instruction

ROtates a register Left.

All bits in register move left

Bits from left-hand end appear on right-hand end, and in the carry flag

To rotate 1 bit:

```
MOV DL,1
```

To rotate more than 1 bit, put number of bits to shift in CL register first.

```
MOV CL,3
ROL BX,CL
```

Figure 6-4 shows how the ROL instruction operates. (In other instructions, called "shifts," the bits pushed off one end of the register disappear forever, and zeros are added at the other end.)
The ROL instruction will rotate any of the registers AX, BX, CX, DX, and also any of the 8-bit halves of these registers: AL, DH, and so on. It will also rotate a memory address.

There are two ways to use this instruction. If you only want to rotate one bit, then you can write:

```
ROL AL,1
```

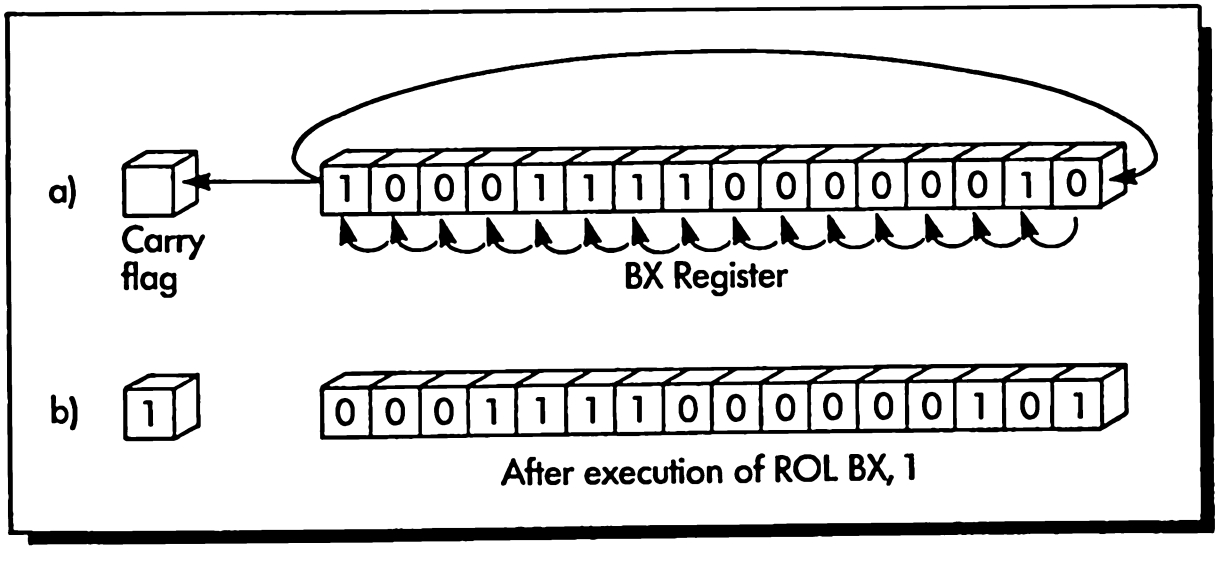


Figure 6-4. Operation of the ROL instruction

(You can use whatever register you want, instead of AL.)

If you want to rotate more than one bit at a time (as we do in BINIHEX), you put *the number of bits you want to rotate in the CL register, before you execute the ROL.* Examples:

```
MOV  CL,4       ;rotate DH four bits left
ROL  DH,CL

MOV  CL,8       ;rotate AX eight bits left
ROL  AX,CL
```

> The CL register acts as a "count" for the number of times to shift a register using a rotate or shift instruction.

The Flags

Before we go on to talk about the next instructions, JNZ, CMP, and JL, you need to be aware of things called "flags." There are nine of these flags, but at this point you don't need to know about all of them. However, there are three or four which are important, and you should be aware of the existence of the others.

The flags are one-bit registers, grouped into a single 16-bit register which is called, logically enough, the flag register. Since there are only nine flags, only nine of the sixteen bits are used, scattered more or less randomly in the register. The reasons for this randomness are historical: The flags in the low part of the register occupy the same bit positions they did in the older 8-bit 8080 microprocessor. The flags in the high

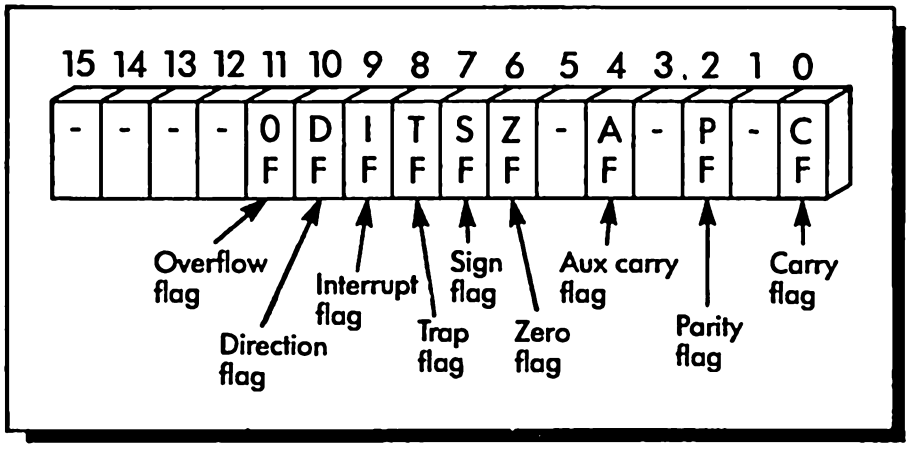


Figure 6-5. The flag register

half of the register are new to the 8088. The flag register is shown in Figure 6-5.

The flags are *set* (meaning set to 1) or *cleared* (meaning set to 0), when certain instructions — mostly those involving comparisons and arithmetic or logical operations — are executed. For instance, if you subtract two numbers, and the result is zero, then the *zero flag* will be automatically set, as in this program fragment:

```
MOV  AL,21
SUB  AL,21
```

That is, if we put 21 in the AL register, and then subtract the same number from it, the result is zero, so the zero flag will be set: that is, it will contain a 1.

Accessing the Flags from DEBUG

It's possible to look at the flags with DEBUG to see how they're set, and to change them if desired. Get into DEBUG and type "R" to see the registers:

```
A>debug
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø8F1:Ø1ØØ Ø3EB          ADD     BP,BX
```

Flags

The two-letter mnemonics on the right in the middle row are the flag settings. The following table (which can be found in the DEBUG section of IBM's *Disk Operating System* manual, under "Register Command") shows what the mnemonics mean. The trap flag is not shown in the DEBUG display, so it is not listed here.

Flag Name	Set	Clear
Overflow (yes/no)	OV	NV
Direction (decrement/increment)	DN	UP
Interrupt (enable/disable)	EI	DI
Sign (negative/positive)	NG	PL
Zero (yes/no)	ZR	NZ
Auxilliary Carry (yes/no)	AC	NA
Parity (even/odd)	PE	PO
Carry (yes/no)	CY	NC

You can change the flag settings in DEBUG by using the RF command. When you type "rf", DEBUG will print out all the flag settings, and wait for you to type in a two-letter mnemonic, which will presumably be the opposite of one of the ones shown.

For instance, if we type "rf" and find that the sign flag is set to PL for "plus," (meaning that bit #7 is 0), then we can set it to NG for "negative" (meaning that the bit will contain 1), by typing "ng", as shown below:

```
-rf
NV UP DI PL NZ NA PO NC -ng
```

Then you can check that the change was made with "r":

```
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØØØ  DX=ØØØØ  SP=FFEE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø8F1  ES=Ø8F1  SS=Ø8F1  CS=Ø8F1  IP=Ø1ØØ   NV UP DI NG NZ NA PO NC
                                                    ↑
                                          Sign flag is now negative
```

The flags we will encounter most often in this chapter are the zero flag, the sign flag, the overflow flag, and the carry flag.

> Flags hold the result of one instruction so another instruction can find out what happened.

Effect of the Flags

Once the flags are set, they can then influence other instructions, usually those called "conditional jumps." For instance, there is an instruction JZ, for "Jump if Zero." If the zero flag is set, this instruction will cause a jump to the address specified in the operand field of the instruction. If the zero flag is *not* set, the instruction following the jump instruction will be executed. We'll examine an example of this process in the BINIHEX program soon, after we discuss a few more instructions.

The DEC Instruction

This instruction is frequently used in counting operations, where a total count is put in a register, and then decremented each time some operation is performed. When the operation has been performed the specified number of times, the count in the register reaches zero, and this sets the zero flag. The zero flag can in turn influence the result of a conditional jump instruction like JNZ.

DEC Instruction

Decrements a register.

Can be used to subtract 1 from any of the 8-bit or 16-bit registers in the 8088.

```
DEC  DX
DEC  SI
DEC  BL
```

Flags affected: AF, OF, PF, SF, ZF

In BINIHEX we want to print four hex digits, so at the start of the program we put the number 4 in the CH register to keep track of this. Then each time we finish printing a digit we decrement the count in CH, using the DEC instruction in location 18. Until CH reaches zero, the JNZ instruction in line 001A (described below) causes a jump back to the "rotate" label at location 0002, where another digit is printed. When CH does become zero, we've finished all four digits, so we go on to the INT 20 instruction in location 001C, which terminates the program. The DEC instruction is half of this process; the other half is JNZ.

The JNZ Instruction

JNZ Instruction

Jumps if zero flag not set. (JNZ stands for "Jump if Not Zero").

Jumps to the memory location in the operand field if the zero flag is not set.

```
JNZ  DO_AGAIN
JNZ  LOC2
```

Note: memory location to be jumped to must be within −128 or +127 bytes from the JNZ instruction.

The mnemonic JNE ("Jump if Not Equal") can also be used for this instruction.

The JNZ instruction is quite straightforward once the zero flag has been set to the appropriate value by some other instruction. If the zero flag is set to zero as a result of a previous arithmetic or logical operation, comparison, or increment or decrement *not* being zero, then the instruction following the JNZ in the program will be executed. If the zero flag is set to one, as the result of a previous operation *being* zero, then the JNZ will cause a jump to the location specified in the operand field of the instruction.

In our program, JNZ will cause a jump back to the "rotate" label, as described above, until the contents of CH becomes zero, at which time the instruction following the JNZ, the INT 20, will be executed.

The CMP Instruction

CMP Instruction

CoMPares two values.

Flags are set according to result of comparison.

To compare two registers:

```
CMP AL,DL
CMP BX,CX
```

To compare a register and an immediate value:

```
CMP AL,111h
CMP CX,1Ø
```

To compare a register with a memory location:

```
CMP CL,MBYTE
CMP DX,MWORD
```

Flags affected: AF, CF, OF, PF, SF, ZF

This instruction compares the values in two registers, and sets the flags according to the results of the comparison. For instance, if two numbers are equal, the zero flag will be set. Also, appropriate flags will be set to show if one number is larger than the other.

Visualizing CMP as Subtraction

One way to visualize what flags are being set by CMP is to imagine that *the second (right-hand) number in the comparison is "subtracted" from the*

first (left-hand) number. We put the word "subtraction" in quotes here because no actual subtraction takes place. The flags are changed *as if* the subtraction had taken place, but nothing is changed in the registers or memory. It's a sort of "phantom" subtraction.

For example, if the AX register contains 200, and the instruction

```
CMP  AX,8Ø
```

is executed, then the sign flag will be set to PL (PLus), since the result of subtracting 80 from 200 is plus. On the other hand, if AX contains 40, and the same instruction is executed, the sign flag will be set to NG (NeGative), since the result of subtracting 80 from 40 is negative. In addition, various other flags will be set, depending on the results of this imaginary subtraction. For instance, if the two numbers being compared by CMP are equal, the zero flag will be set, as if one number had been subtracted from another, leaving zero.

Don't forget that no *actual* subtraction takes place when this instruction is executed. The content of the registers used in the operand field remains the same; only the flags are changed.

The JL Instruction

JL Instruction

Jumps if X is less than Y where X and Y are the operands in a preceding CMP instruction. (JL stands for "Jump if Less than.")

Jumps to the memory location in the operand field if the sign flag is not equal to the overflow flag.

```
CMP  AX,8ØØØh
JL   DO_AGAIN     ;jump if AX less than 8ØØØh

CMP  CL,DL
JL   LOC2         ;jump if CL less than DL
```

Note: memory location to be jumped to must be within −128 or + 127 bytes from the JL instruction.

The mnemonic JNGE ("Jump if Not Greater nor Equal") can also be used for this instruction.

There are two ways to look at the operation of this instruction. One is the official way, shown in the *IBM Personal Computer MACRO Assembler* manual. This states that this instruction will cause a jump only if the sign flag is not equal to the overflow flag. This may be true, but it's not the way you probably want to look at it when you're writing a program. Who knows when the sign flag will equal the overflow flag?

A more useful way to visualize the operation of JL is as the direct result of a CMP instruction. For example, suppose we had the following program fragment:

```
CMP  AL,BL
JL   PURPLE
```

This is equivalent to saying, "If AL is less than BL, then jump to location PURPLE." We read the two items in the CMP statement from left to right as if they were in plain English, and place between them the inequality suggested by the jump instruction, in this case "Less than." This is shown in Figure 6-6.

Thus if AL contains 10 and BL contains 20, the jump to PURPLE will take place, since AL is less than BL; whereas if AL contained 100 and BL contained 20, the jump would not take place.

> The CMP instruction and conditional jump instructions (like JL and JG) work together to form program branches.

In the BINIHEX program we compare the ASCII character in AL with 3Ah, using the CMP instruction at location 000C, in order to find out if the digit to be printed needs to have 7 added to it, as discussed earlier. If AL is less than 3A, then the JL instruction in location 000E causes a jump to PRINTIT. If, however, the content of AL is greater

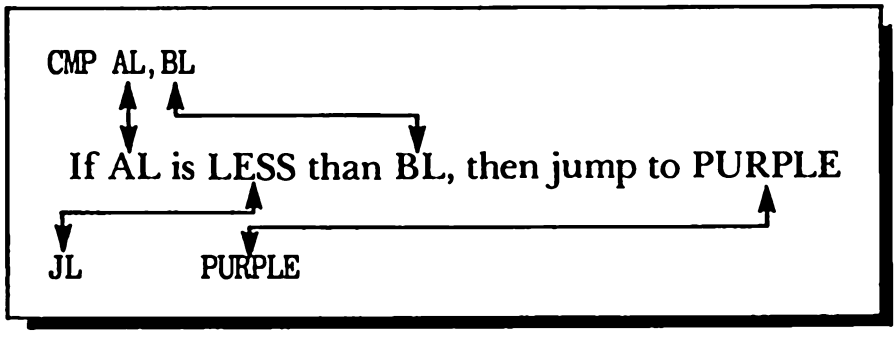


Figure 6-6. Comparisons and inequalities

than or equal to 3A, we go on to the ADD instruction in location 0010, which adds a 7 to the ASCII value of our character.

Using DEBUG's Trace Command

Sometimes (perhaps "usually" is a better word) something goes wrong in the operation of a program that you're writing, and it's difficult to discover what it is simply by examining the listing. When you execute the program everything happens too fast to watch, of course. It would be nice if there were a way to execute a program one instruction at a time, with time between the instructions to see what the effect of each instruction had been. DEBUG's "T" (for "Trace") command does just this.

The operation of "T" is very simple: when you type "t", DEBUG will execute one instruction in your program. Type "t" again, and it will execute the next instruction. Each time it does this it also prints out the contents of all the registers, just as the "R" command does. The instruction which will be executed is the one at the address contained in the IP register, so that by changing the contents of IP (with the "RIP" command) you can start tracing through a program anywhere you like. Once the program is started, of course, the IP is incremented automatically to the next instruction, just as it is when the program is running normally.

Tracing BINIHEX

The program we have just described, BINIHEX, provides a nice example of the use of the trace command. Call up DEBUG and BINIHEX.COM at the same time:

```
A>debug binihex.com
```

You'll need to do a couple of things before you can start tracing. The first is to put a number into the BX register so you can watch the operation of the program as it prints it out. It's good if all the digits in the number are different, so you can distinguish them more easily:

```
-rbx
BX ØØØØ
:1234
```

Now, it is a sad but true fact that you cannot use DEBUG to trace the operation of the DOS function calls. If you try to do it (and you probably will), you'll find that you're tracing into all sorts of strange and wonderful

places in your computer's memory, but that you never get back to your program! The way to avoid this problem is to follow a simple rule: *never try to trace an INT instruction.*

The NOP Instruction

The easiest way to avoid tracing an INT instruction is to put NOP (for "NO oPeration") instructions into our program in place of the INT 21 at line 0016. Because INT 21 requires two bytes, and NOP only one, we need to insert two NOPs, at 0016 and 0017.

```
-a116
Ø9Ø5:Ø116 nop      <— Enter first "nop"
Ø9Ø5:Ø117 nop      <— Enter second "nop"
Ø9Ø5:Ø118          <— Press Enter
-
```

(If you are using DOS version 1, you can use "E" to insert "90" into these two locations. 90 is the machine-language op-code for NOP.)

NOP Instruction

Does nothing at all. (NOP stands for "No OPeration")

Occupies one byte. Useful for replacing unwanted instructions.

NOP is an instruction which does absolutely nothing. However, it takes up one byte of space in memory. Thus it is useful when you want to get rid of some instructions in memory without disturbing the rest of the program.

Operating the Trace Command

Now that we've taken care of the preliminaries, we can start tracing. We'll assume your IP is still set at 0100, as it was when you first loaded DEBUG. If not, change it with RIP.

To see the contents of the registers before the first instruction is executed in trace mode, you should start off your tracing session by typing "r":

```
        Note contents of BX
-r              |
AX=ØØØØ  BX=1234  CX=ØØ1E  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØØ B5Ø4          MOV     CH,Ø4   ↑
                                        |
                                  Start location
```

There's the 1234 you put into the BX register. CL contains 1E, left over from some previous operation, and AX and DX are empty. The IP is at 100, where it should be, and the instruction we are about to execute is MOV CH,04. Let's do it. Type "t" and press ENTER.

```
                Here's the 4 in CH
-t                  |
AX=ØØØØ  BX=1234  CX=Ø41E  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø2   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø2 B1Ø4          MOV     CL,Ø4
```

Great — there's the 04 in CH, the high part of CX. Next we'll put 04 in CL:

```
                  Another 4 in CL
-t                  |
AX=ØØØØ  BX=1234  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø4   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø4 D3C3          ROL     BX,CL
```

And there it is. Now we're going to rotate the BX register left 4 bits, since 4 is the number in CL. Watch what happens to BX:

```
        BX has been rotated
-t              |
AX=ØØØØ  BX=2341  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø6   OV UP DI PL NZ NA PO CY
Ø9Ø5:Ø1Ø6 8AC3          MOV     AL,BL
```

Look at that! The 1 moved around to the right-hand side, and the other three digits shifted over to the left.

For the moment the only part of BX we need is the 1, which is the first digit we're going to print out, so we copy BL to AL:

```
-t
AX=ØØ41  BX=2341  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø8   OV UP DI PL NZ NA PO CY
Ø9Ø5:Ø1Ø8 24ØF          AND     AL,ØF
```

Mask off the extraneous 4 with an AND instruction:

```
   Mask off the 4
-t
AX=ØØØ1  BX=2341  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØA   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØA Ø43Ø          ADD     AL,3Ø
```

Add 30h to make it an ASCII character:

```
   ASCII value
-t
AX=ØØ31  BX=2341  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØC   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØC 3C3A          CMP     AL,3A
```

Now we need to find out whether the hex digit we're printing is from 0 to 9 or from A to F, so we compare it with 3Ah.

```
-t
AX=ØØ31  BX=2341  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØE   NV UP DI NG NZ AC PO CY
Ø9Ø5:Ø1ØE 7CØ2          JL      Ø112
                                        Note change in sign flag
```

Notice that CMP doesn't change any of the registers, but it does change the sign flag from PL (plus) to NG (negative). That's how the JL instruction knows that the contents of AL (31h) is less than 3Ah. So it performs the jump to 0112 (rather than simply going on to the next instruction):

```
-t
AX=ØØ31  BX=2341  CX=Ø4Ø4  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø112   NV UP DI NG NZ AC PO CY
Ø9Ø5:Ø112 8ADØ          MOV     DL,AL
                                  Note that the IP register is changed to new address
  New address
```

Now we're getting set up to print the digit out, using the Display Output function. We MOVe the character from AL into DL, and the code for the Display Output function, which is 2, into AH.

```
                                        ASCII "1"
-t                                          |
AX=ØØ31  BX=2341  CX=Ø4Ø4  DX=ØØ31  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø114   NV UP DI NG NZ AC PO CY
Ø9Ø5:Ø114 B4Ø2           MOV     AH,Ø2
```

We've NOPed out the actual INT instruction, so we pass over the two NOPs without incident:

```
Display Output function number
-t |
AX=Ø231  BX=2341  CX=Ø4Ø4  DX=ØØ31  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø116   NV UP DI NG NZ AC PO CY
Ø9Ø5:Ø116 9Ø             NOP


-t
AX=Ø23Ø  BX=2341  CX=Ø4Ø4  DX=ØØ3Ø  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø117   NV UP DI NG NZ AC PE CY
Ø9Ø5:Ø117 9Ø             NOP
```

Now we want to find out if we've done all four digits, so we decrement CH:

```
-t
AX=Ø23Ø  BX=2341  CX=Ø4Ø4  DX=ØØ3Ø  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø118   NV UP DI NG NZ AC PE CY
Ø9Ø5:Ø118 FECD           DEC     CH
```

This does not change the zero flag from NZ to ZR, since the result in CH is 03; so we jump back to print another digit, starting at location 0102:

```
                  Decremented number
-t                      |
AX=Ø23Ø  BX=2341  CX=Ø3Ø4  DX=ØØ3Ø  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø11A   NV UP DI PL NZ NA PE CY
Ø9Ø5:Ø11A 75E6           JNZ     Ø1Ø2                        |
                                                          Unchanged
```

And here we go again:

```
-t
AX=Ø23Ø  BX=2341  CX=Ø3Ø4  DX=ØØ3Ø  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø2   NV UP DI PL NZ NA PE CY
Ø9Ø5:Ø1Ø2 B1Ø4           MOV     CL,Ø4
                                    |
                          New location after jump
```

You can trace through the process again, and watch the next digit get printed (not really printed, because of the NOPs). We'll leave you to do it on your own, but by this time you should have gotten the idea of how trace works. You will also have gotten a very graphic explanation of how the program works.

The DECIBIN Program

In this section we're going to write another program, called DECIBIN, which accepts a decimal number typed on the keyboard and converts it to a binary number in the BX register. Later we'll combine this program and BINIHEX to create a decimal to hex conversion program. DECIBIN will also be used in several other situations later in the book, where we want to input decimal numbers into a program. In higher-level languages like BASIC this sort of conversion is built in, but in assembly language we need to build up a library of subroutines which we can plug into other programs as we need them. (A good place to find such subroutines is the *Bluebook of Assembly Language Routines for the IBM PC and XT*, by Christopher L. Morgan [New York: Plume/Waite, New American Library, 1984]. It contains more sophisticated versions of the programs we're demonstrating here, as well as many other routines for a wide variety of programming functions.)

Type in the right-hand columns of the following LST file:

```
                        ;DECIBIN--Program to get decimal digits
                        ;        from keyboard and convert them
                        ;        to binary number in BX

0000                    prognam segment

                                assume cs:prognam

0000  BB 0000                   mov     bx,0    ;clear BX for number

                        ;Get digit from keyboard, convert to binary
0003                    newchar:
0003  B4 01                     mov  ah,1       ;keyboard input
0005  CD 21                     int  21h        ;call DOS
0007  2C 30                     sub  al,30h     ;ASCII to binary
0009  7C 10                     jl   exit       ;jump if < 0
000B  3C 09                     cmp  al,9d      ;is it > 9d ?
000D  7F 0C                     jg   exit       ;yes, not dec digit
000F  98                        cbw  ;byte in AL to word in AX
                        ;(digit is now in AX)
```

```
                                ;Multiply number in BX by 1Ø decimal
ØØ1Ø  93                                xchg ax,bx      ;trade digit & number
ØØ11  B9 ØØØA                           mov  cx,1Ød     ;put 1Ø dec in CX
ØØ14  F7 E1                             mul  cx         ;number times 1Ø
ØØ16  93                                xchg ax,bx      ;trade number & digit

                                ;Add digit in AX to number in BX
ØØ17  Ø3 D8                             add  bx,ax      ;add digit to number
ØØ19  EB E8                             jmp  newchar    ;get next digit

ØØ1B                            exit:
ØØ1B  CD 2Ø                             int  2Øh

ØØ1D                            prognam ends

                                        end
```

Then assemble, link, and convert it to a COM file in the same way as before.

Operating the DECIBIN Program

To use this program you execute it, then type in any positive decimal number less than 65535, and then press ↵ (or any key other than a decimal digit). The program will take the decimal number you have typed in, 4096 for example, and convert it to its binary equivalent stored in the BX register. You can then examine the BX register with DEBUG to make sure that the program has done what it's supposed to do.

Since the output of the program is a number in the BX register, there is no point in operating the program directly from DOS. It would work, but you wouldn't be able to see the results. Thus, you must execute the program from DEBUG. However, there is a slight problem: If you load the program in with DEBUG, type "g" to run the program, and then look at the BX register with the R command, you will find that it is always 0000. This is because when the INT 20 function terminates a program, it sets *all* the registers to 0000.

Breakpoints

The answer to this problem is to stop the program before it reaches the INT 20. To do this, we'll use another of DEBUG's features, called *breakpoints*. A breakpoint is a marker you put in a program which says to DEBUG: "Execute the program normally, but when you get to this point, stop the program and print out the contents of the registers, then go back

to DEBUG." DEBUG does this by inserting instructions into the program that will cause a jump out of the program to DEBUG at the specified point, and then immediately replacing those instructions with the original ones from the program once the point is reached.

You set up breakpoints at the same time you execute a program using the "G" command. To do this you simply type the address where you want the breakpoint after the "g" and before you press ↵. You can specify up to ten breakpoints at once this way, but in our case we only need one. We want to stop execution just before we perform the final INT 20 instruction, so we'll put our breakpoint right on top of the INT 20, which is at location 001B. So to run the program we do this:

```
A>debug decibin.com      <— Load in DEBUG and the program
-g 11b                   <— Enter "g" and the breakpoint
65535                    <— Type the decimal number, press ↵
```

Once you hit ↵, the program will be executed; but instead of terminating normally, it will be interrupted at the breakpoint, and DEBUG will print out the registers at that point.

```
AX=Ø1DD  BX=FFFF  CX=ØØØA  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø11B   NV UP DI NG NZ NA PE CY
Ø9Ø5:Ø11B CD2Ø          INT     2Ø
```

BX contains FFFF, which is hex for 65535d, so it works! If we want to try it on another number, we must be careful to set the IP register back to 100, since after the breakpoint IP will retain the address of the breakpoint, not the start of the program:

```
-rip
IP Ø11B
:1ØØ
-g 11b
9
AX=Ø1DD  BX=ØØØ9  CX=ØØØA  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø11B   NV UP DI NG NZ NA PE CY
Ø9Ø5:Ø11B CD2Ø          INT     2Ø
```

Here we typed 9, and got 9 in the BX register.

How Does DECIBIN Work?

The operation of this program makes use of the following algorithm ("algorithm" is a fancy word for "method"):

1. Put zero in BX register.
2. Get decimal digit from keyboard, convert to binary.
3. Multiply whatever was in BX before, by 10d (0Ah).
4. Add new digit to BX.
5. Go back to step 2, unless a non-digit was typed, in which case, the program is finished.

This works because each time we type a decimal digit (instead of ↵), we're really saying two things. First, we're giving the program the value of the new digit in the one's column. Second, we're telling the program that all the digits we typed *before* must be moved one column left; that is, multiplied by 10d.

The 8088, bless its little heart, has a multiply instruction, so we can use that to multiply by 10d, instead of the convoluted algorithms necessary with lesser microprocessors.

There are a few new instructions in this program, including the multiply, so let's examine them before we explore how the program works in detail.

The SUB Instruction

SUB Instruction

Subtracts right-hand operand from left-hand. Result (difference) is stored in left-hand operand.

To subtract contents of registers:

```
SUB AL,BL
SUB BX,CX
```

To subtract number from register:

```
SUB DL,2Ah
```

To subtract register from memory:

```
SUB MWORD,DX
```

Also, a number can be subtracted from a memory address, and a memory address can be added to a register.

Flags affected: AF, CF, OF, PF, SF, ZF

This instruction is rather similar to ADD. You need to remember that the quantity in the right-hand operand is subtracted from the quantity in the left-hand operand (not the other way around.) In DECIBIN we need to subtract 30h from the ASCII code for the character that was typed in, in order to convert it to binary. This is done with the SUB AL,30h in line 0007.

Notice that following this subtraction the flags are set just as if we had compared two numbers with a CMP instruction. So the JL EXIT in the next line will cause a jump if the contents of AL are less than 30h. The only way this could be true is if the character we typed in was not a number at all, so we exit the program on characters less than 30h.

The JG Instruction

JG Instruction

Jumps if X is Greater than Y where X and Y are the operands in a preceding CMP instruction.

Jumps to the memory location in the operand field if the sign flag is equal to the overflow flag and the zero flag is not set.

```
CMP  AX,8ØØØh
JG   DO_AGAIN   ;jump if AX greater than 8ØØØh
CMP  CL,DL
JL   LOC2       ;jump if CL greater than DL
```

Note: memory location to be jumped to must be within −128 or +127 bytes from the JG instruction.

The mnemonic JNLE ("Jump if Not Less nor Equal") can also be used for this instruction.

Flags affected: none

As you can see, JG can be thought of as the opposite of JL. Like JL, it can be interpreted in two different ways. You can think of it as causing a jump when the sign flag is equal to the overflow flag and when the zero flag is 0. Or you can use the more intuitive approach and think of it as jumping when the left-hand operand in a preceding CMP (or SUB)

instruction is greater than the right-hand operand. (See the discussion on the JL instruction.) Figure 6-7 shows how this works.

In our DECIBIN program we've changed the ASCII digit that was typed in, to binary. Now we want to check if it's greater than 9, since if it is, it is not a decimal digit after all and we want to exit from the program. So we compare AL with 9d in line 000B, and if AL is greater than 9d, we jump to exit, via the JG EXIT instruction in line 000D.

The CBW Instruction

CBW Instruction

Converts Byte to Word.

The byte must be in AL, the word is always in AX.

If the number in AL is positive, AH is filled with 00.

If the number in AL is negative, AH is filled with FF.

```
CBW
```

Flags affected: none

This is a useful instruction when you've been dealing with an 8-bit quantity (a byte), and you want to make it into a 16-bit quantity (a word). Later on in the program we're going to add the binary digit in the A register to the binary number in BX, and we need both these registers to

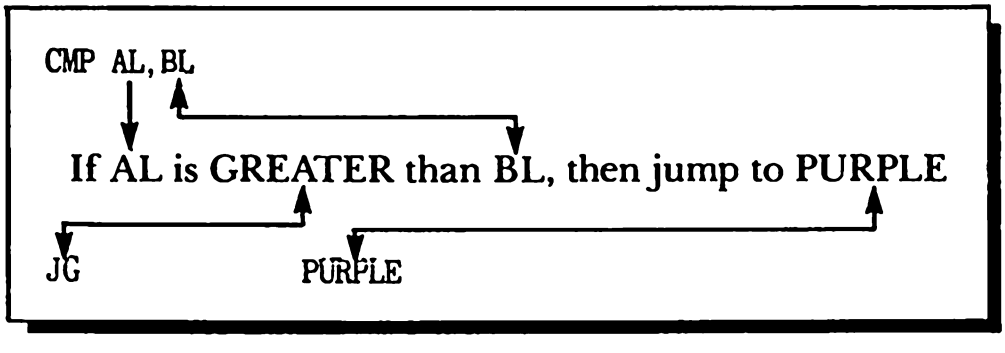


Figure 6-7. Comparisons and inequalities with JG

be words, since the ADD instructions can not add a byte to a word. We start off with a byte in AL, so to make it into a word in AX, we use the CBW instruction.

Note that assuming the number in AL is positive (bit 7 is zero), AH will be automatically set to zero by this instruction.

The XCHG Instruction

XCHG Instruction

Exchanges the contents of two registers, or a register and a memory location.

Works on either 8-bit or 16-bit registers (the segment registers cannot be used).

```
XCHG  AX,BX
XCHG  CL,AL
XCHG  MWORD,DX
XCHG  BL,MBYTE
```

Flags affected: none

In order to multiply two words, one must be in AX and the other in some other register (as we'll see). We have a number in BX we want to multiply by 10, and we have the digit we're going to add to it later in AX. The easiest way to handle this is to switch them: exchange AX and BX. Then we put the 10d into CX, and we're ready to multiply AX by CX. After we're done, we switch AX and BX back again with another XCHG instruction, so the effect has been to multiply BX by 10. This process is shown in Figure 6-8.

The MUL Instruction

In our program the 16-bit word is switched from BX into AX, and multiplied by 10d, which leaves the result, a huge 32-bit quantity, in DX and AX. We are not interested in the high half of the result, in DX, since we are not going to try to convert numbers larger than FFFF anyway. So

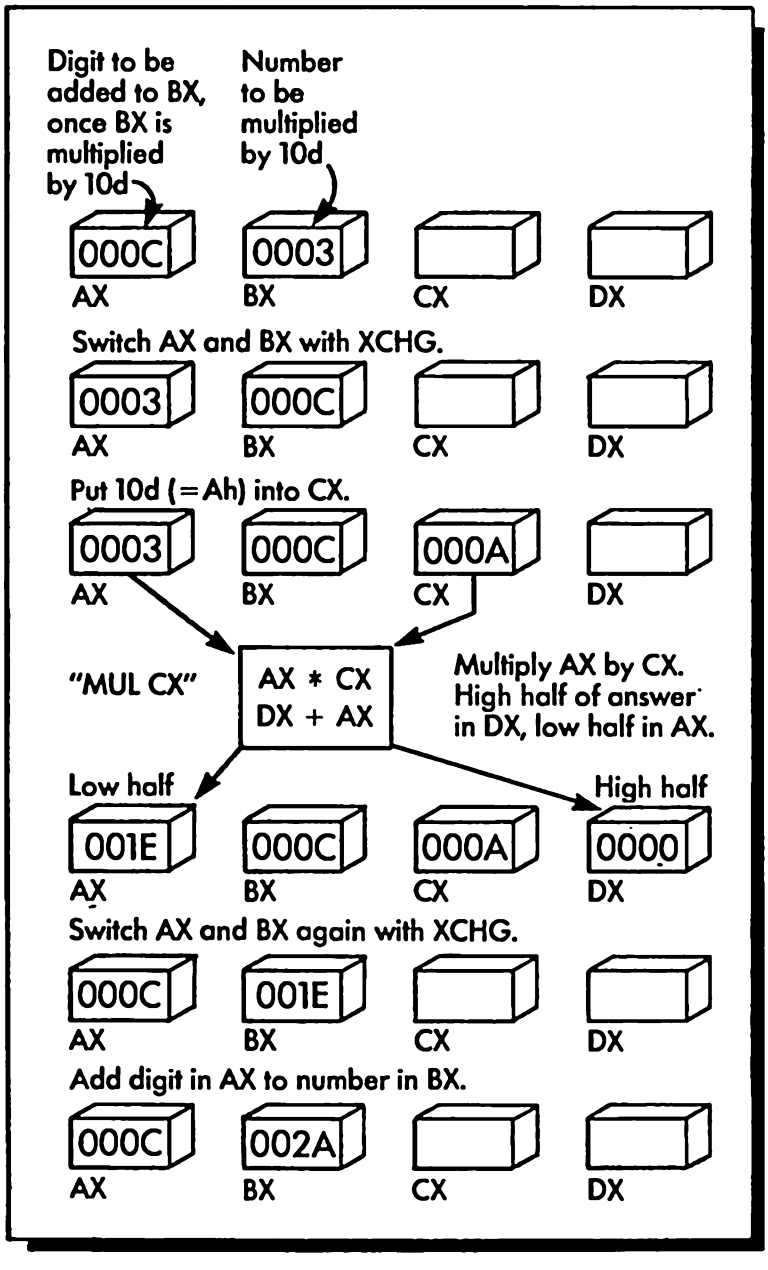


Figure 6-8. Operation of DECIBIN program

we ignore DX, and switch the result in AX back into BX. Then we add the digit (which started in AX, switched to BX, and is now back in AX). The binary equivalent of the decimal digits which have been typed in so far is now in BX, and we go back to read another character, which will either be another digit, or a nondigit which will cause us to exit from the program.

MUL Instruction

MULtiplies contents of A register, and operand register or memory address.

To multiply bytes, one number is in AL, second is in 8-bit register, or in memory:

```
MUL CL
MUL BL
MUL MBYTE
```

Result is a 16-bit quantity in AX.

To multiply words, one number is in AX, second is in 16-bit register, or in memory.

```
MUL CX
MUL BX
MUL MWORD
```

Result is a 32-bit quantity, high half in DX, low half in AX.

Flags affected: CF and OF = 0 if high-order half of result is zero, otherwise they = 1.

The DECIHEX Program

We're now ready to combine BINIHEX and DECIBIN into a veritable giant of a program called DECIHEX, for "Decimal to Hexadecimal" converter. This program will use DECIBIN to get a decimal number from the keyboard and convert it to binary in the BX register, and then BINIHEX to print out the contents of BX on the screen in hex. Our plan is to take DECIBIN and BINIHEX and modify

them slightly so they are both *procedures* instead of programs. (We'll explain procedures soon.) Then we'll write a short main program which will call each procedure in turn.

We'll need another small addition to our program: a routine to print a carriage return (cr) and linefeed (lf). We need to print the cr/lf combination after we have received the decimal number from the user; otherwise the hex number we print out — the hex equivalent of the number — will print on top of the original decimal number on the screen display.

The overall structure of the DECIHEX program is shown in Figure 6-9.

As you can see, there's one main program and three procedures. The main program calls the three procedures in turn.

Here's the complete program:

```
                        ;DECIHEX--Main Program
                        ;  Converts decimal on keybd to hex on screen

                        ;*********************************************
0000                    decihex segment
```

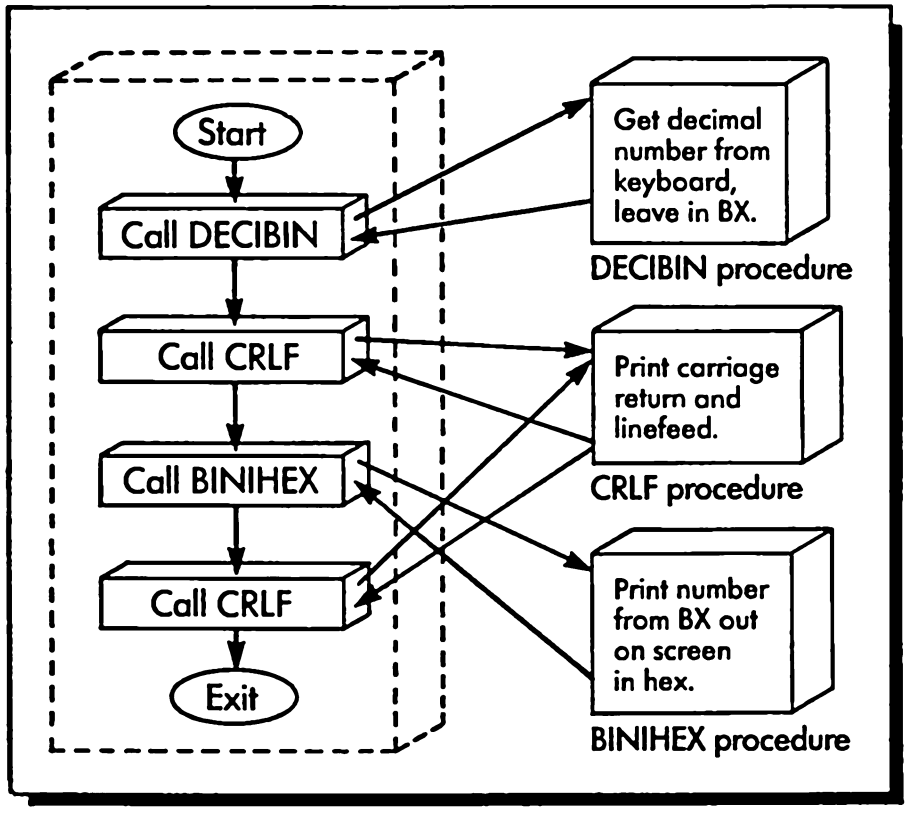


Figure 6-9. Structure of the DECIHEX program

```
                                assume  cs:decihex

                        ;MAIN PART OF PROGRAM.  Connects procedures
                        ;  together.

0000  E8 000E R         repeat: call    decibin ;keyboard to binary
0003  E8 0047 R                 call    crlf    ;print cr and lf

0006  E8 002A R                 call    binihex ;binary to screen
0009  E8 0047 R                 call    crlf    ,print cr and lf

000C  EB F2                     jmp     repeat  ;do it again

                        ;-----------------------------------------
                        ;PROCEDURE TO CONVERT DEC ON KEYBD TO BINARY
                        ;  Result is left in BX register

000E                    decibin proc    near

000E  BB 0000                   mov     bx,0    ,clear BX for number

                        ;Get digit from keyboard. convert to binary
0011                    newchar:
0011  B4 01                     mov  ah,1       ;keyboard input
0013  CD 21                     int  21h        ;call DOS
0015  2C 30                     sub  al,30h     ;ASCII to binary
0017  7C 10                     jl   exit       ;jump if < 0
0019  3C 09                     cmp  al,9d      ;is it > 9d ?
001B  7F 0C                     jg   exit       ;yes. not dec digit
001D  98                        cbw  ;byte in AL to word in AX
                        ;(digit is now in AX)

                        ;Multiply number in bx by 10 decimal
001E  93                        xchg ax,bx      ;trade digit & number
001F  B9 000A                   mov  cx,10d     ;put 10 dec in CX
0022  F7 E1                     mul  cx         ;number times 10
0024  93                        xchg ax,bx      ;trade number & digit

                        ;Add digit in ax to number in bx
0025  03 D8                     add  bx,ax      ;add digit to number
0027  EB E8                     jmp  newchar    ;get next digit
0029                    exit:
0029  C3                        ret             ;return from decibin

002A                    decibin endp            ;end of decibin proc
                        ;-----------------------------------------
                        ;PROCEDURE TO CONVERT BINARY NUMBER IN BX
                        ;  TO HEX ON CONSOLE SCREEN
```

```
ØØ2A                    binihex proc    near

ØØ2A  B5 Ø4                     mov  ch,4       ;number of digits
ØØ2C  B1 Ø4             rotate: mov  cl,4       ;set count to 4 bits
ØØ2E  D3 C3                     rol  bx,cl      ;left digit to right
ØØ3Ø  8A C3                     mov  al,bl      ;move to AL
ØØ32  24 ØF                     and  al,Øfh     ;mask off left digit
ØØ34  Ø4 3Ø                     add  al,3Øh     ;convert hex to ASCII
ØØ36  3C 3A                     cmp  al,3ah     ;is it > 9 ?
ØØ38  7C Ø2                     jl   printit    ;jump if digit = Ø to 9
ØØ3A  Ø4 Ø7                     add  al,7h      ;digit is A to F
ØØ3C                    printit:
ØØ3C  8A DØ                     mov  dl,al      ;put ASCII char in DL
ØØ3E  B4 Ø2                     mov  ah,2       ;Display Output funct
ØØ4Ø  CD 21                     int  21h        ;call DOS
ØØ42  FE CD                     dec  ch         ;done 4 digits?
ØØ44  75 E6                     jnz  rotate     ;not yet

ØØ46  C3                        ret             ;return from binihex

ØØ47                    binihex endp

                        ;-----------------------------------------
                        ;PROCEDURE TO PRINT CARRIAGE RETURN
                        ;         AND LINEFEED

ØØ47                    crlf    proc    near

ØØ47  B2 ØD                     mov  dl,Ødh     ;carriage return
ØØ49  B4 Ø2                     mov  ah,2       ;display function
ØØ4B  CD 21                     int  21h        ;call DOS

ØØ4D  B2 ØA                     mov  dl,Øah     ;linefeed
ØØ4F  B4 Ø2                     mov  ah,2       ;display function
ØØ51  CD 21                     int  21h        ;call DOS

ØØ53  C3                        ret             ;return from crlf

ØØ54                    crlf    endp

                        ;-----------------------------------------
ØØ54                    decihex ends
                        ;*****************************************

                                end
```

There it is, by far the largest program you've worked on to date. You should be able to save a lot of typing by using your word-processing program to merge DECIBIN.ASM and BINIHEX.ASM together, and

then adding the other parts of the program to them.

You'll notice that only one change has been made to the instructions in DECIBIN and BINIHEX: the INT 20 instruction at the end of each has been changed to a RET. That's because INT 20 is *only* used to return from a main program to DOS or DEBUG. It doesn't work at all to return from a procedure, which is what we need to do at the end of DECIBIN, BINIHEX, and LFCR.

Type in the program in the usual way, assemble it, link it, and convert it to a COM file. As before, we'll tell you what it does before we talk about how it works.

Operating the DECIHEX Program

Your diligence in writing DECIBIN and BINIHEX can now be rewarded. Our new DECIHEX program does not require DEBUG to operate. You can use it from DEBUG if you want, but you can also invoke it directly from DOS.

Once it's loaded it will sit there waiting for you to type in a decimal number. Type in the number, up to 65535d, then press (↵). The hex equivalent will be printed out on the next line. (Don't try negative numbers: the program can't handle them. It will exit if you type any character except the decimal digits 0 through 9.)

```
A>decihex          <— Enter name of program
4Ø96               <— Enter decimal number
1ØØØ               <— Hexadecimal result will be displayed
1Ø                 <— Enter another decimal number
ØØØA               <— Hex result printed out
^C                 <— Type (Ctrl) (Break) to exit
A>                 <— Back in DOS
```

The program will then wait for another number, and so on. To escape from the program you'll need to type (Ctrl) C or (Ctrl) (Break), since no escape mechanism was built into the program itself.

How Does DECIHEX Work?

The main part of the program consists entirely of CALL instructions. These, along with the RETs at the end of the procedures, are the only new instructions in the program. There is also a new pseudo-op, called PROC. These three things are interrelated, so let's see what they do.

CALL Instruction

CALLs a subroutine.

Transfers control to the address of the subroutine in the operand field.

Also sets up return by placing address following the CALL on the stack.

CALL can be either short or long.

In a short CALL, the contents of the IP register are placed on the stack.

In a long CALL, the contents of, first the CS register, and then the IP register, are placed on the stack.

```
CAL SUBR
```

Flags affected: none

A CALL is like a JMP to another memory location, except that in addition to jumping to a new location, the CALL instruction also stores the memory address of the location just following the CALL instruction itself. Where does it store this address? In a part of memory called the *stack*. We're not going to get into the operation of the stack at this point; it will be covered in the next chapter. For the time being you can simply think of it as a place to store addresses. The result is that when a RET instruction is executed at the end of a procedure, the 8088 knows what memory address to return to. The operation of CALL and RET in procedures is diagrammed in Figure 6-10.

There are two variations of the CALL instruction. Long CALLs are calls to procedures in a *different* memory segment than the calling program. Since we haven't learned about segments yet, we'll ignore this possibility for the moment. Short CALLs are made to procedures in the *same* segment as the calling program, which is the case in our program. Now, the tricky part is that, although a long CALL and a short CALL use different machine language op-codes, *there is no difference in the way long calls and short calls are written in the source (ASM) file.* They are both just CALL. How then does the assembler know, when it sees CALL, whether to assemble a long call or a short one?

The answer is that it looks at the routine you have CALLed to get its answer. And what exactly is it that you CALL? It's something called a *procedure.* A procedure is a group of assembly language instructions, much like a program, which have been grouped together. A procedure usually performs a specific, well-defined task. In BASIC and some other higher-level languages, and in other dialects of assembly language, a procedure is often called a *subroutine.*

The PROC Pseudo-Op

The PROC pseudo-op is used to identify procedures. Remember the SEGMENT and ENDS pseudo-ops that were used to define a segment? The PROC pseudo-op is similarly part of a *pair*: PROC and ENDP. They're used to surround a procedure, like this:

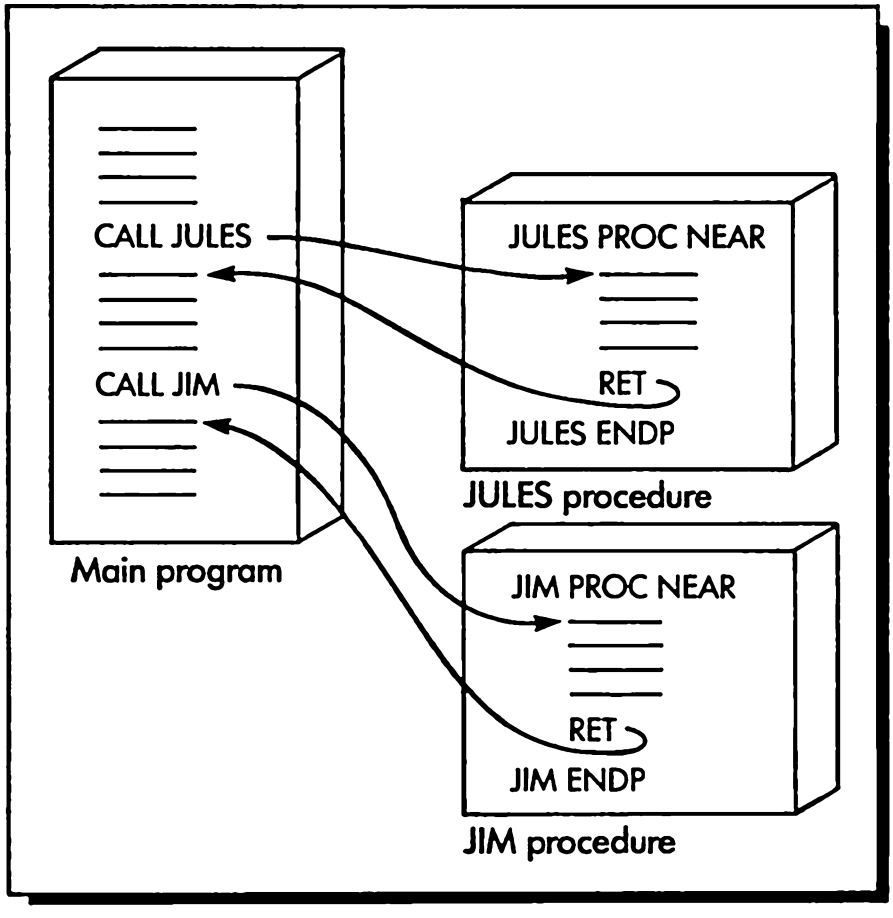


Figure 6-10. Operation of the CALL and RET instructions

```
SUB_NAME  PROC     FAR   ;start of procedure
```

(procedure goes here)

```
          RET            ; (procedure ends with RET)
SUB_NAME  ENDP           ;end of procedure
```

There are two kinds of "procedures," NEAR and FAR. These definitions have to do only with how the procedure will be called with CALL and how it will return to the calling program with RET. If it is a NEAR procedure, it will be called with a NEAR CALL, and if it is a FAR procedure it will be called with a FAR CALL. So the only real function of the PROC pseudo-op (besides setting off blocks of code and so making the documentation of the programming a little clearer), is to tell the assembler whether CALLs to that procedure will be NEAR or FAR.

The RET Instruction

RET Instruction

RETurns from procedure.

Transfers control to the address on the top of the stack. This address was placed there earlier by a CALL instruction.

RET can be either near or far.

A near RET returns from a NEAR procedure, taking one word from the stack and placing it in the IP.

A far RET returns from a FAR procedure, taking the first word from the stack and placing it in the IP, then taking the second and placing it in CS.

```
RET
```

Optionally, RET can also pop additional values off the stack.

```
RET 12
```

Flags affected: none

The RET instruction transfers control from a procedure back to the program which called it. This is possible because the address of the instruction following the CALL instruction is stored on the stack. When the 8088 sees the RET, it simply looks on the stack, and transfers control to the address it finds there. RET instructions can be NEAR or FAR. This is determined by the assembler, which looks to see if the RET is in a NEAR or FAR procedure. If it's in a NEAR procedure, the RET is assembled as a NEAR RET. If it's a FAR procedure, then it's a FAR RET.

PROCs, CALLs, and RETs

So, as we've seen, all the "nears" go together and all the "fars" go together. A NEAR CALL calls a NEAR PROC which returns with a NEAR RET, and a FAR CALL calls a FAR PROC which returns with a FAR RET. When we're writing our program the only thing we need to specify is the PROC: we must choose FAR or NEAR. The assembler will figure out the CALLs and the RETs.

The idea behind this way of doing things is to make it harder to make a mistake. If *you* had to specify both the CALLs and the RETs, the chances are you would sooner or later mix them up, and then your program would be in big trouble. Using the PROC approach means that the *assembler* has the responsibility of matching up the CALLs and RETs, and it, presumably, is infallible.

> PROC was invented to save you from yourself.

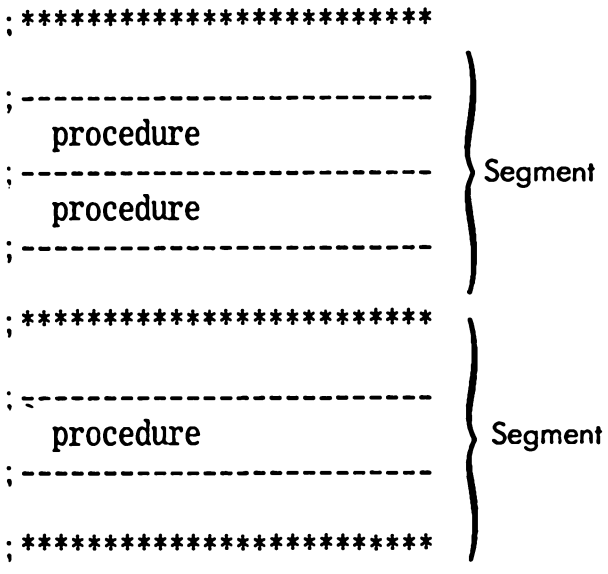
Since we are only dealing with calls in the same code segment at this point, all our PROCs, CALLs and RETs will be NEAR.

The operation of the DECIHEX program should now be clear. The four CALL instructions at the start of the program spell it out: get a decimal number from the keyboard with DECIBIN, then print a cr/lf to move down to the next line; print the hex version of the number with BINIHEX, then print another cr/lf to get ready for the next input.

Note that DECIBIN leaves the binary equivalent of the decimal number in BX. It stays there, unchanged, because LFCR does not damage the contents of BX. Then, when BINIHEX is called, it finds this number in BX and converts it to hex. We can say that BX has been used to *pass* (transfer) a value from one procedure to another.

Formatting Procedures and Segments

To highlight the various procedures in our program listing, we have divided them with dashed lines. This makes it easier to see at a glance where all the procedures begin and end in a long program. Similarly, we've started the *segment* with a line of stars (asterisks), and ended it with another line of stars. Although only one segment is used in this program, future programs will have more segments, and visually separating one from another will make the listing easier to read. We'll follow these conventions throughout the book. The general structure of this format is shown below.



Cross-Reference: Using the CREF Program

Before leaving this chapter, we're going to introduce you to another program: CREF. This program is interesting, but certainly not essential to the operation of the assembler and the writing of the short assembly language programs in this book. If you're in a hurry, you can skip this section. But if you plan to write really long assembly language programs at some future time, then read on.

The CREF program is included on the disk with MASM and ASM, and is described in the *IBM Personal Computer MACRO Assembler* manual. The purpose of CREF is to produce a cross-reference listing of the symbolic names used in the program.

What good is a cross-reference listing? In short programs it isn't all that valuable. But when you're debugging a really long program it can be

very useful. Suppose you find, for instance, that you need to change the symbolic name of a particular location in memory from REPEAT to DO_AGAIN. It's easy to change the REPEAT location itself, but what about all the references to it? You may have all sorts of JMP REPEAT instructions scattered throughout your program. The cross-reference file generated by CREF gives you an easy way to find where they all are.

Line Numbers and the PAGE Pseudo-Op

As we'll see below, the CREF program generates its cross-reference table using the *line numbers* in the program listing to refer to various locations. These line numbers are simply the ordinal number of each line in the listing, starting at the top. However, as we've seen, no such numbers appear on the LST files generated by ASM that we've seen so far in this book. You could count the lines yourself on the ASM or LST file, but this is rather tedious; it's the kind of thing computers are supposed to do for you.

It turns out there is a way to get the assembler to generate these line numbers: the PAGE pseudo-op. PAGE is used at the beginning of an ASM file, mostly to specify the number of characters per line and the number of lines per page in the LST file. One instance in which this might be useful is if you have a printer with 132 columns, and you want the LST file to make use of this increased width. In our case we aren't going to change the width so we use the PAGE pseudo-op without any parameters. This gives us default values of 66 lines per page and 80 characters per line. This is just what you get if you don't use PAGE at all. However, *using PAGE causes the line-numbering feature to be turned on.*

The DECIHEX program that follows is somewhat different from the previous DECIHEX program we examined. The beginning of the ASM file of this DECIHEX program shows how the PAGE pseudo-op is positioned at the start of the program.

```
        page                    ←— Page pseudo-op

:DECIHEX--Main program
:  Converts decimal on keybd to hex on screen
:
decihex segment
;-----------------------------------------
:
main    proc    far
:
        assume  cs:decihex
```

```
;
;MAIN PART OF PROGRAM.  Links procedures
;  together.
;
display equ     2h      ;video output
key_in  equ     1h      ;keyboard input
doscall equ     21h     ;DOS interrupt number
;
        push    ds      ;ds on stack
        sub     ax,ax   ;set ax=Ø
        push    ax      ;zero on stack
;
repeat: call    decibin ;keyboard to binary
        call    lfcr    ;print lf and cr
;
. . . . . (balance of program deleted) . . . . .
```

Once you have PAGE in your ASM file, you can assemble the file and generate the cross-reference file. When you use the assembler you need to specify the CREF filename, as shown here:

```
A>asm decihex
The IBM Personal Computer Assembler
Version 1.ØØ (C)Copyright IBM Corp 1981

Object filename [DECIHEX.OBJ]: nul        ←— OBJ file not needed
Source listing  [NUL.LST]: decihex        ←— Specify the LST filename
Cross reference [NUL.CRF]: decihex        ←— Specify the CRF filename

Warning Severe
Errors  Errors
Ø       Ø
```

The first output file from the assembler that we're interested in is the LST file, shown below. As you can see, it now sports line numbers on the left-hand side.

Unfortunately, these line numbers take up a lot of room, so that the comments on the right get chopped off on our 80-column screen and on our 80-column printer. This is a rather serious problem. We don't want to sacrifice comments in order to have line numbers. The best solution is to have a printer wider than 80 columns. In any case, here's the beginning of the LST file, with the comments cut off on the right to fit on the page:

```
1                                       page
2                               ;DECIHEX--Main program
3                               ;  Converts decimal on keybd to hex on
4                               ;
5       0000                    decihex segment
6                               ;-----------------------------------
7                               ;
8       0000                    main    proc    far
9                               ;
10                                      assume  cs:decihex
11                              ;
12                              :MAIN PART OF PROGRAM.  Links subrouti
13                              ;  together.
14                              ;
15      = 0002                  display equ     2h      ;video output
16      = 0001                  key_in  equ     1h      ;keyboard inpu
17      = 0021                  doscall equ     21h     ;DOS interrupt
18                              ;
19      0000  1E                        push    ds      ;ds on stack
20      0001  2B C0                     sub     ax,ax   ;set ax=0
21      0003  50                        push    ax      ;zero on stack
22                              ;
23      0004  E8 0012 R         repeat: call    decibin ;keyboard to b
24      0007  E8 004B R                 call    lfcr    ;print lf and
25                              ;
26      000A  E8 002E R                 call    binihex ;binary to scr
27      000D  E8 004B R                 call    lfcr    ;print lf and
28                              ;
29      0010  EB F2                     jmp     repeat  ;do it again
30                              ;
31      0012                    main    endp
32                              ;-----------------------------------
33                              ;
34      0012                    decibin proc    near
35                              ;
36                              ;PROCEDURE TO CONVERT DEC ON KEYBD TO
37                              ;  result is left in BX register
38                              ;
39      0012  BB 0000                   mov     bx,0    ;clear BX for
40                              ;
41                              ;get digit from keyboard, convert to b
```

. . . . . (balance of program deleted) . . . . . .

The assembler also generated a CRF file. This is an intermediate step in the generation of the cross-reference file, which has the file extension REF. The CREF program is used to generate the REF file from the CRF file. (What would a linguist make of a sentence like that?)

```
A>cref decihex                    <— Enter CREF and filename
List [DECIHEX.REF]:               <— Specify the name of the REF file
```

The resulting REF file is shown below. The line number where a symbol is *defined* is marked with a number sign (#). The other numbers are all the other line numbers in the program where that symbol is referenced. Thus REPEAT occurs in line 23, and is referenced only in line 29 (as you can verify from the LST file above). As we noted, for short programs the use of line numbers and the CREF utility is of somewhat questionable value. However, if you are writing really long programs, especially if you have a 132-column printer, it can be very useful.

```
Symbol Cross Reference                  (# is definition)      Cref-1
BINIHEX ..............................  26     67#    89

DECIBIN ..............................  23     34#    64
DECIHEX ..............................   5#    10     106
DISPLAY ..............................  15#    83     95     99
DOSCALL ..............................  17#    44     84     96    100

EXIT .................................  46     48     61#

KEY_IN ...............................  16#    43

LFCR .................................  24     27     92#    103

MAIN .................................   8#    31     107

NEWCHAR ..............................  42#    60

PRINTIT ..............................  79     81#

REPEAT ...............................  23#    29
ROTATE ...............................  73#    86
```

Summary

In this chapter you've honed your skills with the MACRO Assembler by practicing on three different programs, ending up with a very usable decimal to hex conversion program. You've learned many new 8088 instructions, and two new features of DEBUG: the T command, and the use of breakpoints. You've also been introduced to PROCedures, CALLs, RETurns, and the relationship among them. Finally you learned about the PAGE pseudo-op and the CREF cross-reference utility.

7

How Does It Sound?

Concepts
White noise
The stack
Using the timer to generate sound
Putting data in memory: why we need segments

8088 Instructions
ROR = Rotate right
PUSH = Store on stack
POP = Remove from stack
NEG = Change sign of number
SHL = Shift left (or SAL = Shift arithmetic left)
DIV = Divide
EQU = Equivalence: (pseudo-op)

DOS Functions
Check Standard Input Status (Check Keyboard Status)

Applications
NOISE program — Sound of surf
GUN program — Machine gun
SIREN program — Strange siren sound
SPACEWARS program — Unearthly burble
KAZOO program — Plays a kind of music
PIANO program — Turns keyboard into piano

You've spent the last two chapters working hard, learning how to use the assembler and writing some pretty serious programs. For a change of pace we're going to write some programs that use the sound-producing capabilities of your PC. These programs will be kind of fun, and they have the advantage that you can *hear* instantly what the program is doing, so you can debug them with your ears!

There will also be some new things to learn about the 8088 and the assembler in this chapter. We'll talk about the mysterious thing called the "stack," and about the EQU pseudo-op, and we'll cover a number of new 8088 instructions. We'll also touch on storing data in memory, which will lead into the next chapter, on memory segmentation.

Why Use Sound?

Why would you want to use sound in your programs? If you're writing a game, of course, the answer is obvious: everybody likes music and sound effects. But even gray flannel suit programs like data base managers and spreadsheets can profit from the use of sound. Most such programs make a limited use of sound to indicate error messages, just as the keyboard routine in your DOS will beep at you when its buffer gets too full. But there is also a lot to be said for programs that play simple tunes to indicate the completion of a process or the need for input from the user, or that make rude noises when you make a mistake. Sound can become part of your interaction with the machine. Just as graphics (which we'll be covering in a later chapter) can aid this interaction, sound can provide a more interesting and varied form of communication between human and computer.

The White Noise Program

Remember our first sound program (called SOUND), the simple tone generator in chapter 2? It looked like this, disassembled with the "U" command:

```
-u1ØØ,1Øe
Ø9Ø5:Ø1ØØ E461          IN      AL,61
Ø9Ø5:Ø1Ø2 24FC          AND     AL,FC
Ø9Ø5:Ø1Ø4 34Ø2          XOR     AL,Ø2
Ø9Ø5:Ø1Ø6 E661          OUT     61,AL
Ø9Ø5:Ø1Ø8 B94ØØ1        MOV     CX,Ø14Ø
Ø9Ø5:Ø1ØB E2FE          LOOP    Ø1ØB
Ø9Ø5:Ø1ØD EBF5          JMP     Ø1Ø4
```

This program created a sound using a very fundamental method: it actually sent signals to the loudspeaker which pushed the cone of the loudspeaker in and out. (You might want to reread this section in chapter 2, if your memory of it has become hazy.)

One of the advantages of this system of sound generation is that it gives you complete control over the speaker. You don't have to send it a fixed note or frequency — you can mix up all sorts of different frequencies and send them at the same time. We're going to take advantage of this possibility to create a kind of sound called "white noise." It is an equal mixture of all audio frequencies, just as white light is a mixture of different frequencies of light, or colors.

If you're a BASIC programmer you'll recognize that there's no way to get the kind of effect we're describing here in that language. You're restricted to the fixed group of musical tones that BASIC wants you to use. Only assembly language gives you the freedom to fool around with the sound generators, achieving strange and wonderful effects, some of which, if you're inventive, may have never been heard on earth before.

Here's the LST file for the NOISE program. Type in the assembly language part of it, and assemble, link, and convert it to a COM file in the usual way. Then try it out. You should get a rushing sound, like static, with no musical tone at all.

Like the earlier SOUND program, the only way to stop this program is with a system reset. Sorry about that — we'll show you how to fix this problem soon.

```
                        ;NOISE--Makes a sound with the
                                speaker
                        ;  can't be stopped except by reset
                        ;
                        ;*********************************************
0100                    prognam segment ;define code segment

                        ;---------------------------------------------
0100                    main    proc    far     ;main part of program

                                assume  cs:prognam

0100                            org     100h    ;start of program

0100                    start:          ;starting execution address

0100  BA 0140                   mov  dx,140h ;initial value of wait

0103  E4 61                     in   al,61h     ;get port 61
0105  24 FC                     and  al,11111100b  ;AND off bits 0, 1
0107  34 02             sound:  xor  al,2       ;toggle bit #1 in AL
0109  E6 61                     out  61h,al     ;output to port 61
```

```
Ø1ØB  81 C2 9248                add  dx,9248h   ;add random pattern
Ø1ØF  B1 Ø3                     mov  cl,3       ;set to rotate 3 bits
Ø111  D3 CA                     ror  dx,cl      ;rotate it

Ø113  8B CA                     mov  cx,dx      ;put in CX
Ø115  81 E1 Ø1FF                and  cx,1ffh    ;mask off upper 7 bits
Ø119  81 C9 ØØØA                or   cx,1Ø      ;ensure not too short

Ø11D  E2 FE             wait:   loop wait       ;wait
Ø11F  EB E6                     jmp  sound      ;keep on toggling

Ø123                    main    endp    ;end of main part of program
                        ;-----------------------------------------
Ø123                    prognam ends    ;end of code segment
                        ;*****************************************

                                end     start   ;end assembly
```

This program is really not so different from the SOUND program of chapter 2. Of course it looks different because it's written in real assembly language, rather than with DEBUG. But the only operational difference is in the length of time we wait between sending pulses to the speaker. In SOUND we always waited the same length of time, an interval determined by the number we placed in the CX register before we executed the LOOP instruction in location 010B. Since these intervals were always the same, the resulting sound was a more or less pure tone.

What would happen if instead of using constant intervals, as in SOUND, we sent pulses to the speaker at varying intervals? We'd get a mixture of all audio frequencies at equal energy, and the result would be white noise. That's what the NOISE program does. We send a pulse, wait a few milliseconds, send another pulse, wait a completely *different* number of milliseconds, send another pulse, and so on.

Random Number Generator

The question is, how do we get some random numbers to use for the intervals between sending the pulses? There are all sorts of ways to do this. The one we've selected is simple, but effective.

We keep our random number in the DX register. Every time we need a new random number, we add a fixed number to the old one (the fixed number is 9248h, which is 1001001001001000 binary), and then rotate the result 3 bits right. These two actions are sufficient to produce a series of numbers which are random enough for our needs in this program. It also uses a new instruction, ROR.

The ROR Instruction

ROR Instruction

ROtates a register Right.

All bits in register move right.

Bits from right-hand end appear on left-hand end, and in the carry flag.

To rotate 1 bit:

```
ROR DL,1
```

To rotate more than 1 bit, put number in CL first:

```
MOV CL,3
ROR BX,CL
```

Flags affected: CF, OF

This instruction is just like the ROL instruction in the last chapter, except that it moves all the bits right instead of left.

Now that we have a random number, we need to make it a little less random. We need to keep it between certain limits: it should not be larger that 200h, nor smaller than 10h. These numbers correspond roughly to the useable frequencies of the speaker.

We make sure our number isn't too large by masking off the upper 7 bits with an AND instruction, so the result has a maximum value of 1FFh. And we make sure it isn't too small by ORing on a 10h, which means it can never be smaller than that. The resulting semi-random number is then placed in the CX register and used as a counter for the LOOP instruction, just as in the old SOUND program. Then we jump back up to toggle bit number 1 in the AL again, and the process is repeated over and over, resulting in the strange noise you heard from your speaker when you ran the program.

What's this noise good for? Well, it's a nice imitation of distant surf, or freeway traffic. A short burst of it would sound like an explosion. And combined with normal tones, all sorts of effects are possible. Our next example will show you how to turn it into a burst of machine gun fire.

The Machine Gun Program

It's not that we want to dwell particularly on instruments of destruction, but this sound is a rather easy one to derive from the preceding program, and it does begin to demonstrate some of the ways we can use white noise. If you prefer, you can think of the sound as coming from a high-speed typist.

Here's a good opportunity to save yourself some keystrokes. Use the COPY command to make a copy of the NOISE program. You can call it GUN. Then modify it with your word processor until it looks like this:

```
                        ;GUN--Makes machine gun sound
                        ;  fires fixed number of shots

                        ;*****************************************
Ø1ØØ                    prognam segment ;define code segment

                        ;-----------------------------------------
Ø1ØØ                    main    proc    far     ;main part of program

                                assume  cs:prognam

                                org     1ØØh    ;start of program

Ø1ØØ                    start:          ;starting execution address

Ø1ØØ  B9 ØØ14                  mov  cx,2Ød     ;set number of shots
Ø1Ø3                    new_shot:
Ø1Ø3  51                        push cx         ;save count
Ø1Ø4  E8 Ø111 R                 call shoot      ;sound of shot
Ø1Ø7  B9 4ØØØ                   mov  cx,4ØØØh   ;set up silent delay
Ø1ØA  E2 FE             silent: loop silent     ;silent delay
Ø1ØC  59                        pop  cx         ;get shots count back
Ø1ØD  E2 F4                     loop new_shot   ;loop till shots done
Ø1ØF  CD 2Ø                     int  2Øh        ;return to DOS

Ø111                    main    endp    ;end of main part of program
                        ;-----------------------------------------
                        ;SUBROUTINE TO MAKE BRIEF NOISE

Ø111                    shoot   proc near

Ø111  BA Ø14Ø                   mov  dx,14Øh    ;initial value of wait
Ø114  BB ØØ2Ø                   mov  bx,2Øh     ;set count

Ø117  E4 61                     in   al,61h     ;get port 61h
Ø119  24 FC                     and  al.111111ØØb ;AND off bits #Ø, #1
```

```
Ø11B  34 Ø2              sound:  xor  al,2        ;toggle bit #1 in AL
Ø11D  E6 61                      out  61h,al      ;output to port 61

Ø11F  81 C2 9248                 add  dx,9248h   ;add random bit pattrn
Ø123  B1 Ø3                      mov  cl,3       ;set to rotate 3 bits
Ø125  D3 CA                      ror  dx,cl      ;rotate it

Ø127  8B CA                      mov  cx,dx      ;put in CX
Ø129  81 E1 Ø1FF                 and  cx,1ffh    ;mask off upper 7 bits
Ø12D  81 C9 ØØØA                 or   cx,1Ø      ;ensure not too short

Ø131  E2 FE              wait:   loop wait       ;wait


                         ;made noise long enough?
Ø133  4B                         dec  bx         ;done enough?
Ø134  75 E5                      jnz  sound      ;jump if not yet

                         ;turn off sound
Ø136  24 FC                      and  al,11111ØØb  ;AND off bits Ø, 1
Ø138  E6 61                      out  61h,al     ;turn off bits Ø, 1

Ø13A  C3                         ret             ;return from subr

Ø13B                     shoot   endp
                         ;-----------------------------------------
Ø13B                     prognam ends    ;end of code segment
                         ;*****************************************

                                 end     start   ;end assembly
```

So how does this all work? As you can see, our old program has been turned into a PROCedure, or subroutine, which is CALLed from the main program. The main program first sets up a count in the CX register. This count is the number of shots to be fired. Each shot will consist of an interval of white noise followed by an interval of silence. So we set up this count, and then call SHOOT — which is the noise subroutine — to make the noise, and then have a moment of silence created by the LOOP SILENT instruction at location 010A. Then we decrement the count of the number of shots, and check to see if this count has gone to zero; if so, we return to DOS; if not, we go do another shot.

But as you can see, it isn't really as simple as that. After all, we start off using the CX register to hold the number of shots, but later we put 4000h into it to cause a silent delay. Why doesn't the count of the number of shots get destroyed when we put the 4000h into CX? The answer is

that we've *saved the contents of CX on the stack,* by using a PUSH instruction. Later we get the contents back from the stack by using a POP instruction.

What is this thing called a stack? And where is it?

The Stack, PUSH, and POP

The stack is a strange and mysterious entity which is very important in the design and programming of most modern computer systems. It is, in fact, often said that a computer uses "stack architecture" if it has a stack — meaning that the stack is the important thing to know about the overall design of the computer.

What is the stack? Operationally, it's a special place to put 16-bit quantities, like the contents of the CX register (or any of the other 16-bit registers). To store the contents of a register on the stack, you execute a PUSH instruction, and to take it off again, you do a POP.

The stack is a lot like any other stack, say a stack of dishes in a restaurant. When a dish is washed and ready to be stored, it's placed on the top of the stack. When a waiter needs a new plate he takes it off the top of the stack. This kind of stack is called LIFO, for "Last In, First Out," meaning that the last dish placed on the stack is the first one to be removed.

Of course, in our stack we're not storing dishes, but 16-bit quantities. You can use the stack yourself to store the contents of registers, but the 8088 also uses the stack. What for? To store the return address when we do a CALL instruction to a procedure. (A NEAR CALL will cause a one-word address to be stored on the stack, and a FAR CALL will store a two-word address, although so far we're only interested in NEAR CALLS.) As you learn more about the stack you'll begin to realize how CALL and RET operate.

> The stack is a place in memory to store the contents of 16-bit registers.

What sort of thing is the stack? It's simply a sequence of memory locations: a section of memory set aside to serve as temporary storage. How does stack storage differ from ordinary memory storage? First, it's faster and easier: the PUSH and POP instructions which store and retrieve values from the stack are short, efficient instructions. Second, since there is only one stack (at any given time, anyway), it's easy to find.

If one routine puts a value on the stack, another routine knows it can get it off the stack if it needs it. All this will be made clearer as we go along.

Where Is the Stack?

Which memory locations does the stack occupy? Well, you can tell where they are simply by using the "R" command in DEBUG. Let's load the GUN program and try it:

```
A>debug gun.com
                                          Contents of stack pointer
-r                                                  |
AX=ØØØØ  BX=ØØØØ  CX=ØØ14  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØØ B914ØØ        MOV     CX,ØØ14
```

There's a register we haven't told you about yet, called the Stack Pointer (SP) register. There it is on the top row, fifth from the left. (If this were a yearbook photo of the registers, the Stack Pointer would probably have been voted "Most Popular," whereas the Instruction Pointer would have been named "Most Likely to Succeed." Well, anyway.)

The purpose of the Stack Pointer is *to point to the top of the stack.* What does this mean? Simply that the SP register contains the address of the 16-bit value which was *last put on the stack.* Thus if the SP register starts

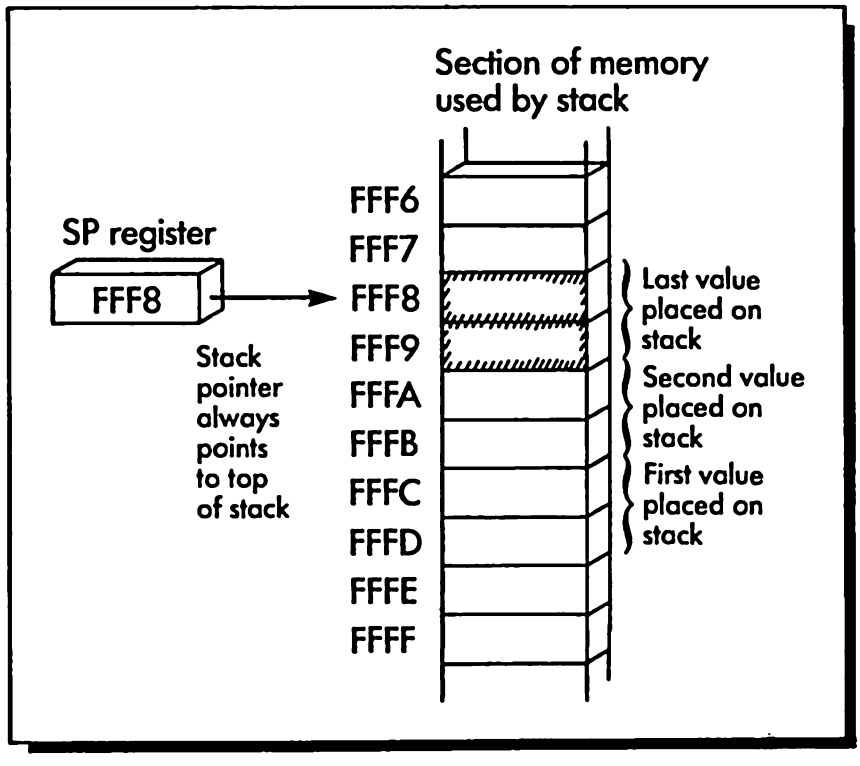


Figure 7-1. Stack pointer register

out at FFFE and you put something on the stack, SP will automatically change to FFFC. After the second thing it will change to FFFA, and after the third thing it will contain FFF8. Figure 7-1 shows how the Stack Pointer looks after three 16-bit quantities have been placed on the stack.

In the DEBUG printout, the top of the stack is at memory location FFFE. How did it get there? It turns out that when a COM file is loaded, the Stack Pointer is automatically set to the top of the memory segment that the program is in. In any given segment the addresses run from

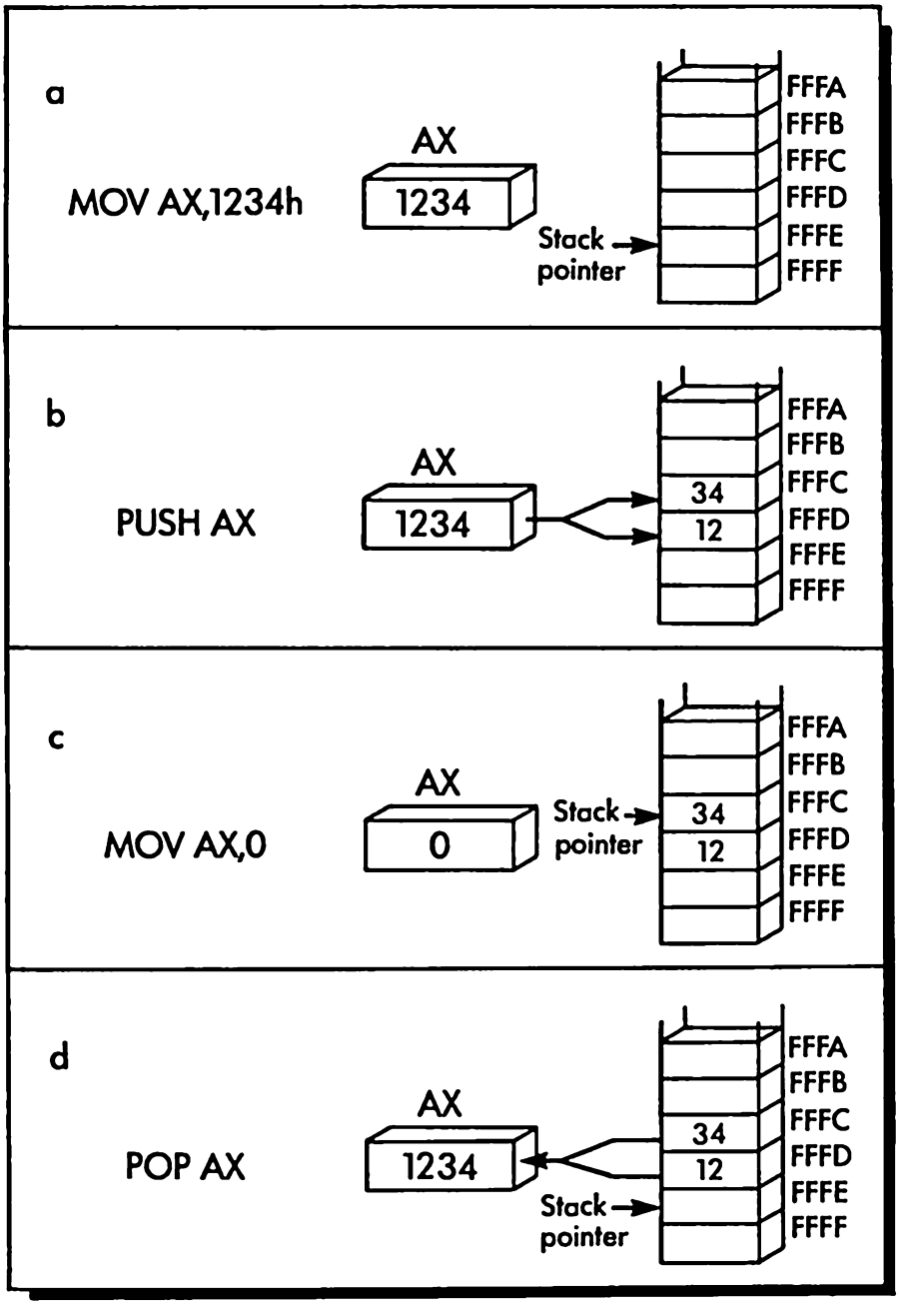


Figure 7-2. Operation of the stack

0000 to FFFF. The program, as you know, starts at 0100 and grows toward higher addresses.

The stack, on the other hand, starts at FFFE and grows in the other direction, toward lower addresses. (It can't start at FFFF because, since it's storing 16-bit quantities, it must always point to *even*-numbered locations.) Figure 7-2 shows a number being stored from the AX register onto the stack with a PUSH, then removed with a POP.

In this figure we show the AX register being loaded with a known quantity: 1234h. The stack pointer is at FFFE. Then we take this quantity in the AX register and PUSH it onto the stack. PUSH AX means that we transfer the 16-bit value from the AX register to the stack, and at the same time change the SP register to point to the address of this value on the stack.

Now, to show that there is nothing up our sleeve, so to speak, we zero out the contents of AX by loading it with 0. Is the original value now lost? No, it's still on the stack. How can we get it back again? With a POP AX instruction, which causes the contents of the memory location pointed to by the Stack Pointer to be transferred from the stack back into the AX register.

The PUSH Instruction

PUSH Instruction

Moves 16-bit quantity from register to stack (or from memory location to stack).

First the SP (Stack Pointer) register is decremented 2 bytes. Then the content of the register (or memory location) in the operand field is written to the stack, at the address pointed to by SP.

```
PUSH AX
PUSH DX
PUSH SI
PUSH ES
PUSH MWORD
```

POP Instruction

Moves 16-bit quantity from stack to register (or from stack to memory location).

First the contents of the stack location pointed to by SP (the Stack Pointer) are transferred into the register (or memory location) in the operand field. Then the address in SP is incremented two bytes.

```
POP  AX
POP  DX
POP  SI
POP  ES
POP  MWORD
```

Let's use DEBUG to get a feel for what's happening when we use the stack, and for how the PUSH and POP instructions work. Bring up DEBUG and type in the following little program. This is not really an executable program, so don't try to run it with "G". It is meant to be traced through one step at a time with "T" to give you a graphic picture of what the stack is doing. Whereas in Figure 7-2 we PUSHed one 16-bit number onto the stack and then POPped it off, we are now going to use DEBUG to show *two* 16-bit quantities being PUSHed onto the stack and POPped off.

```
-a1ØØ
Ø8F1:Ø1ØØ mov ax,1234
Ø8F1:Ø1Ø3 mov bx,5678
Ø8F1:Ø1Ø6 push ax
Ø8F1:Ø1Ø7 push bx
Ø8F1:Ø1Ø8 mov ax,Ø
Ø8F1:Ø1ØB mov bx,Ø
Ø8F1:Ø1ØE pop bx
Ø8F1:Ø1ØF pop ax
Ø8F1:Ø11Ø
```

Here it is, unassembled with "U":

```
-u1ØØ,1Øf
Ø9Ø5:Ø1ØØ B83412        MOV     AX,1234
Ø9Ø5:Ø1Ø3 BB7856        MOV     BX,5678
Ø9Ø5:Ø1Ø6 5Ø            PUSH    AX
Ø9Ø5:Ø1Ø7 53            PUSH    BX
Ø9Ø5:Ø1Ø8 B8ØØØØ        MOV     AX,ØØØØ
Ø9Ø5:Ø1ØB BBØØØØ        MOV     BX,ØØØØ
Ø9Ø5:Ø1ØE 5B            POP     BX
Ø9Ø5:Ø1ØF 58            POP     AX
```

This program loads AX with 1234h, BX with 5678h, and then puts the contents of these two registers on the stack, using PUSH instructions. Then it puts zeros in AX and BX so we can see they're really empty (nothing up our sleeve!). Finally it POPs the old values back off the stack into the registers again. An important thing to notice here is that the order of POPping things *off* the stack must be the *reverse* of the order they were PUSHed *on* with.

> POP things off the stack in the opposite order they were PUSHed on.

Let's follow the operation of our test program, step by step. Enter "R" to see the registers before we start tracing. The first two instructions load the registers with the constants. Note how the contents of AX and BX change.

```
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØ1Ø  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØØ B83412        MOV     AX,1234
-t

AX=1234  BX=ØØØØ  CX=ØØ1Ø  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø3   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø3 BB7856        MOV     BX,5678
-t

AX=1234  BX=5678  CX=ØØ1Ø  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø6   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø6 5Ø            PUSH    AX
```

Now we PUSH the contents of AX and BX onto the stack. Notice how the Stack Pointer changes from FFFE to FFFC (2 fewer than FFFE), and then to FFFA (2 fewer again). The Stack Pointer is *decremented* because the stack is growing *downward* in memory, from higher to lower addresses. It moves two bytes at a time because we are storing 16-bit quantities, which require two bytes each.

```
-t
AX=1234  BX=5678  CX=ØØ1Ø  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø7   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø7 53            PUSH    BX
-t

AX=1234  BX=5678  CX=ØØ1Ø  DX=ØØØØ  SP=FFFA  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø8   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø8 B8ØØØØ        MOV     AX,ØØØØ
```

Now we zero out AX and BX:

```
-t
AX=ØØØØ  BX=5678  CX=ØØ1Ø  DX=ØØØØ  SP=FFFA  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØB   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØB BBØØØØ        MOV     BX,ØØØØ

-t
AX=ØØØØ  BX=ØØØØ  CX=ØØ1Ø  DX=ØØØØ  SP=FFFA  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØE   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØE 5B            POP     BX
```

Fine — both registers are zeroed out. (Remember that the POP at 010E above has not yet been executed: the printout shows the instruction *about to be* executed.) Let's look at the area of memory where the stack is, to see what's happening.

```
          Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
-dffcØ,ffff
Ø9Ø5:FFCØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø9Ø5:FFDØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø9Ø5:FFEØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ  ................
Ø9Ø5:FFFØ  ØØ ØØ ØA Ø1 ØE Ø1 Ø5 Ø9-ØØ Ø6 78 56 34 12 ØØ ØØ  ..........xV4...
```

BX stored here

AX stored here

The first PUSH instruction stored the content of the AX register, which was 1234, into memory locations FFFD and FFFC (remember, 16-bit quantities are always stored most significant byte first; that's why the numbers appear to be backwards). The location to be stored into is always 1 less than the address in the Stack Pointer. The PUSH instruction also changes the Stack Pointer from FFFE to FFFC.

The next PUSH instruction stores the contents of BX, 5678, into memory locations FFFB and FFFA, and changes the Stack Pointer to FFFA.

When we take things off the stack with POP instructions the process is reversed. 16-bit quantities are moved from their memory locations on the stack, and placed in the register specified in the operand field of the POP. The Stack Pointer is also *incremented* two bytes for each POP. Thus, although the quantities previously placed on the stack are actually still in memory, they are no longer accessible to PUSH and POP instructions. They have been "forgotten" because the Stack Pointer is now pointing above them in memory, and the next time something is placed on the stack, they will be written over.

Let's see how POP works:

```
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØ1Ø  DX=ØØØØ  SP=FFFA  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØE   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØE 5B            POP     BX
-t

AX=ØØØØ  BX=5678  CX=ØØ1Ø  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØF   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØF 58            POP     AX
-t

AX=1234  BX=5678  CX=ØØ1Ø  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø11Ø   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø11Ø 9Ø             NOP
```

Each POP places the appropriate value in the specified register, and we're back where we started. (The last instruction, NOP, is not executed.)

Conclusion of the Machine Gun Program

Now we can use our newly acquired knowledge of the stack to understand the operation of the machine gun program. In fact, let's use the same technique we did above: we'll trace through it with "T" to see how it works.

```
A>debug gun.com

-r
AX=ØØØØ  BX=ØØØØ  CX=ØØ14  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØØ   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØØ B914ØØ        MOV     CX,ØØ14
```

We put the number of shots (20d is 14h) in CX, then we save it on the stack with a PUSH CX.

```
-t
AX=ØØØØ  BX=ØØØØ  CX=ØØ14  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø3   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø3 51            PUSH    CX
-t

AX=ØØØØ  BX=ØØØØ  CX=ØØ14  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø4   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø4 E8ØAØØ        CALL    Ø111
```

We don't want to trace the operation of the SHOOT subroutine (which starts at 0111), so we'll skip over the CALL to it by advancing the IP register to one instruction past the CALL, which is address 0107. Then we'll type "R" to see where we are:

```
-rip
IP Ø1Ø4         <— Old value of IP, ready to do CALL
:1Ø7            <— New value of IP, one byte past the CALL
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØ14  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø7   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø7 B9ØØ4Ø        MOV     CX,4ØØØ
```

Now we set up a delay loop by placing 4000h in CX, and then executing the LOOP instruction 4000h times.

```
-t
AX=ØØØØ  BX=ØØØØ  CX=4ØØØ  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØA   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØA E2FE          LOOP    Ø1ØA
```

However, we don't really want to trace through this loop 4000h times, so again we skip over it by manually resetting the IP register to the instruction following the LOOP:

```
-rip
IP Ø1ØA          ←— Old IP value: location of loop
:1Øc             ←— New IP value: one byte past the loop
-r
AX=ØØØØ  BX=ØØØØ  CX=4ØØØ  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØC   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØC 59            POP     CX
-t

AX=ØØØØ  BX=ØØØØ  CX=ØØ14  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1ØD   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1ØD E2F4          LOOP    Ø1Ø3
```

When we execute the POP CX instruction, guess what happens? The value of the number of shots to fire, which was 14h, is restored in the CX register. Great! That's exactly what we wanted. Now we can do the LOOP instruction to see whether we've done all the shots yet.

```
-t
AX=ØØØØ  BX=ØØØØ  CX=ØØ13  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø9Ø5  CS=Ø9Ø5  IP=Ø1Ø3   NV UP DI PL NZ NA PO NC
Ø9Ø5:Ø1Ø3 51            PUSH    CX
```

And we haven't, so we wind up at the beginning of the main program again, saving the CX register on the stack — and the whole process starts again. It will only terminate when we've used up all the shots: that is, when the count in CX goes to zero.

Generating Sound with the Timer

Thus far we have been generating sounds by toggling the speaker on and off with individual instructions from our program. This technique gives us the maximum control of the kinds of sounds we want to make. However, there is another way to generate sounds, one which is more convenient, but provides less control. The remaining programs in this chapter will make use of this second technique, which is to use what IBM calls a "timer."

This timer is an oscillator circuit which is built into the computer. It does to the speaker just what your SOUND program from chapter 2 did: It sends a series of evenly spaced pulses to the speaker. However, it does this with hardware rather than software. Figure 7-3 shows how the hardware is organized.

There are actually three different timers, or oscillators, built into the PC. Timer0 is used in DMA (Direct Memory Access) data transfers, and needn't concern us here. Timer1 is used as the system clock. It causes an

interrupt 18 times every second to update the time. Timer2 is connected to the speaker.

Using the timer to generate sounds is a little more complicated than simply sending pulses to the speaker, as we did before. There are three steps involved. First we load a certain number, 10110110 binary, into timer2 to "initialize" it. Second we load a 16-bit number into timer2 to establish the frequency of the tone to be generated. Finally we open a switch in output port 61h to actually turn on the sound. The program fragment below shows how this is done.

```
;load 1/pitch into timer2 (assume it's in BX)

       mov   al,1Ø11Ø11Øb  ;put magic number
       out   43h,al        ;     into timer2
       mov   ax,bx         ;move 1/pitch into AX
       out   42h,al        ;LSB into timer2
       mov   al,ah         ;MSB to AL, then
       out   42h,al        ;  to timer2

;turn on tone
       in    al,61h        ;read port B into AL
       or    al,3          ;turn on bits Ø and 1
       out   61h,al        ;to turn on speaker
```

The number that we send the timer is not the frequency (pitch) of the tone we want, but a number proportional to the *inverse* of the frequency. That is, the larger the number we send, the lower the pitch will be. Numbers above 2000h generate very low tones, while those smaller than

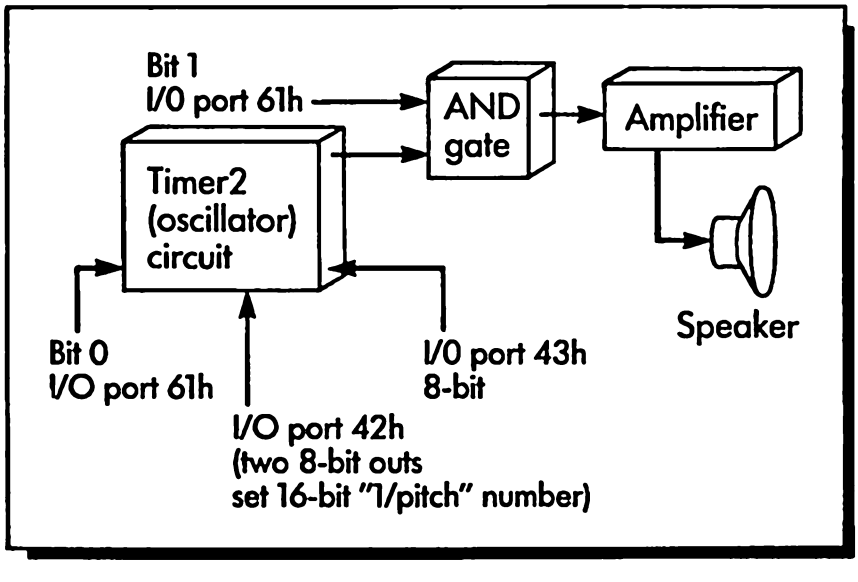


Figure 7-3. Sound hardware

about 10h generate tones too high for human hearing. We'll refer to the number we send the timer as 1/pitch, so we won't forget that it's not the frequency. Figure 7-4 shows the relationship of our "1/pitch" number and the frequency.

Let's talk in more detail about the three steps needed to use timer2 as a sound generator. First we alert the timer that we're about to send the number. We do this by sending a special binary number, 10110110. The reasons for the various bit settings in this number are too complex to go into here — all we need to know is that the number does its job, getting the timer ready to receive the 1/pitch number.

Since 1/pitch is a 16-bit number, and the input and output ports operate with 8 bits, we need to send the number in two steps: the least significant byte (LSB) followed by the most significant byte (MSB). The OUT instruction uses the AL register, so we must juggle the MSB from AH to AL with a MOV instruction.

Turning on the gate to start the tone sounding is just the same as it was in the earlier examples using the bit-toggling approach. Now, however, once we turn on the sound, the tone will continue until we specifically turn it off. Our program doesn't need to continue to interact with the speaker — it can go off and do whatever it wants, and the sound will continue. In fact, once we turn on the sound this way, it will continue until we reset the computer, or send instructions from our program to stop it. This can be especially useful in game programs, where we want a sound to continue while the program does something else: updating the video display, for example.

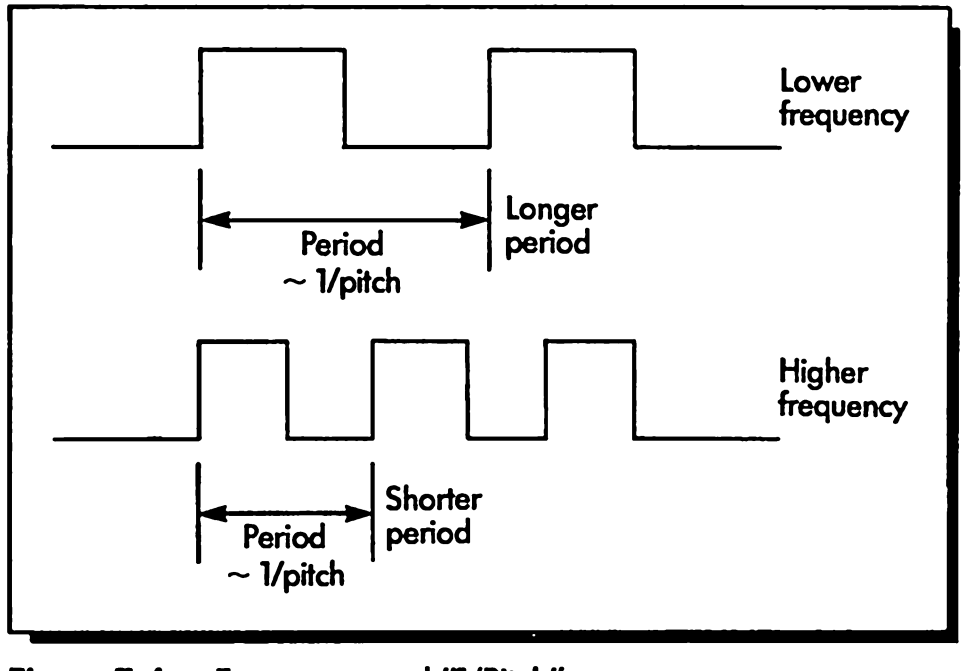


Figure 7-4. Frequency and "1/Pitch"

The SIREN Program

This program is a simple one which makes use of the timer to produce a variable pitch. In fact, it produces all possible pitches, by starting with a 1/pitch number of FFFFh (the lowest possible tone) and going all the way up to 0, the highest possible tone. This is done by keeping the 1/pitch number in the BX register, and decrementing it every so often. Each time we establish a new, higher pitch we wait in the LOOP instruction for a while to give the tone a chance to sound; then we go back to lower the pitch again. Notice that once the magic number 10110110b is installed in the timer, it's not necessary to replace it each time we send the 1/pitch.

```
                        ;SIREN--Uses Timer2 to run speaker
                        ;  features siren effect
                        ;
                        ;*********************************************
Ø1ØØ                    prognam segment ;define code segment

                        ;---------------------------------------------
Ø1ØØ                    main    proc    far     ;main part of program

                                assume  cs:prognam

                                org     1ØØh    ;first address


Ø1ØØ                    start:          ;starting execution address

                        ;start 1/pitch at FFFFh
Ø1ØØ  BB FFFF                   mov   bx,Øffffh ;set 1/pitch in BX

                        ;sound the tone
Ø1Ø3  BØ B6                     mov   al,1Ø11Ø11Øb  ;put magic number
Ø1Ø5  E6 43                     out   43h,al    ;     into timer2
Ø1Ø7  8B C3             tone:   mov   ax,bx     ;move 1/pitch into AX
Ø1Ø9  E6 42                     out   42h,al    ;LSB into timer2
Ø1ØB  8A C4                     mov   al,ah     ;MSB to AL, then
Ø1ØD  E6 42                     out   42h,al    ;  to timer2
Ø1ØF  E4 61                     in    al,61h    ;read port B into AL
Ø111  ØC Ø3                     or    al,3      ;turn on bits Ø and 1
Ø113  E6 61                     out   61h,al    ;to turn on speaker

                        ;increase the pitch and wait a bit
Ø115  4B                        dec   bx        ;increase the pitch
Ø116  B9 ØØ64                   mov   cx,1ØØd   ;set up wait loop
Ø119  E2 FE             wait:   loop  wait      ;wait
Ø11B  EB EA                     jmp   tone      ;go do new tone
```

```
Ø11D                    main    endp    ;end of main part of program
                        ;-----------------------------------------
                         ;
Ø11D                    prognam ends    ;end of code segment
                        ;*****************************************
                        ;
                                end     start   ;end assembly
```

Most of the program is just like the code fragment shown earlier. We've added instructions to set BX to FFFF at the beginning. After the tone has been turned on, we decrement BX to get ready to set the next tone, and then wait with the LOOP for the tone to sound for a while at that pitch.

Try typing the program in, assembling, linking, converting and running it. You'll find that the lower pitches rise very slowly, but that as the pitch gets higher the rate of change increases, shooting up suddenly into inaudibility.

You can modify this program in various ways to change the sound. Instead of starting at FFFFh and going to 0, you could narrow the limits, and thus the frequency shift of the number. By decreasing the number placed in the CX register for the delay, you can speed up the program, and at the same time start to introduce a new "voice," a change in the quality of the sound.

Again, the only way to exit from this program is to reset your computer. However, this is the last time we will require this inelegant method of terminating a program. We promise!

The Space Wars Program

Our next program is the same as SIREN except for two things. First, instead of jumping back up to simply place a new value for 1/pitch into the timer, we now jump two instructions higher, to "sounder," and reset the timer with the magic number. This causes a significant change in the sound quality, as you will see when you run the program.

Second, we have incorporated a new feature — you can now terminate the program and cause a return to DOS by pressing any key while the program is running. Hooray! No more tedious groping for the on-off switch.

Here's what the program looks like:

```
;SPACEWARS--Uses Timer2 to run speaker
;  produces weird rising burble
```

```
                        ;*******************************************
Ø1ØØ                    prognam segment ;define code segment

                        ;-------------------------------------------
Ø1ØØ                    main    proc    far     ;main part of program

                            assume cs:prognam
                            org 1ØØh ;starting address
Ø1ØØ                    start:          ;starting execution address

                        ;initial value of 1/pitch
Ø1ØØ  BB Ø2ØØ                   mov   bx,2ØØh   ;set 1/pitch in BX

                        ;sound the tone
Ø1Ø3                    sounder:
Ø1Ø3  BØ B6                     mov   al,1Ø11Ø11Øb  ;put magic number
Ø1Ø5  E6 43                     out   43h,al    ;     into timer2
Ø1Ø7  8B C3             tone:   mov   ax,bx     ;move 1/pitch into AX
Ø1Ø9  E6 42                     out   42h,al    ;LSB into timer2
Ø1ØB  8A C4                     mov   al,ah     ;MSB to AL, then
Ø1ØD  E6 42                     out   42h,al    ;  to timer2
Ø1ØF  E4 61                     in    al,61h    ;read port B into AL
Ø111  ØC Ø3                     or    al,3      ;turn on bits Ø and 1
Ø113  E6 61                     out   61h,al    ;to turn on speaker

                        ;increase the pitch and wait a bit
Ø115  4B                        dec   bx        ;increase the pitch
Ø116  74 E8                     jz    start     ;when BX=Ø, reset it

Ø118  B9 Ø32Ø                   mov   cx,8ØØd   ;set up wait loop
Ø11B  E2 FE             wait:   loop  wait      ;wait

                        ;check if keyboard character typed
Ø11D  B4 ØB                     mov   ah,Øbh    ;get kbd status
Ø11F  CD 21                     int   21h       ;call DOS
Ø121  FE CØ                     inc   al        ;if AL not FF, then
Ø123  75 DE                     jnz   sounder   ;  no key pressed

                        ;key pressed, return to DOS
Ø125  CD 2Ø                     int   2Øh       ;return
                                               to DOS
Ø127                    main    endp    ;end of main part of program
                         ;--------------------------------
0127                    prognam ends    ;end of code segment
                        ;*******************************************
                                end     start   ;end of assembly
```

Check Standard Input Status DOS Function

To check to see if a key was pressed, we use a new DOS function, "Check Standard Input Status."

CHECK STANDARD INPUT STATUS Function — Number 0Bh

Enter with:

```
Reg AH = 0bh
```

Execute with:

```
INT 21h
```

Return with:

```
AL=FFh if character typed
AL=ØØh if nothing typed
```

Comment: Ctrl Break causes exit from function.

This command was called "Check Keyboard Status" in DOS versions 1.00 and 1.10, and generally speaking this would be an appropriate name, and certainly a less ponderous one. However, in DOS version 2.00 there is *redirection* of input and output, so that sometimes this function will be used with devices other than the keyboard. Hence the substitution of "Standard Input" for "Keyboard." However, for our purposes it's the same thing.

The feature of this function that makes it especially useful is that it doesn't *wait* for you to type something. It checks the keyboard: If something is typed it puts FF in the AL register; if nothing is typed it puts 00 in AL. In either case it goes on to the next instruction in the program. This makes it an indispensable function in those situations where we want our program to keep running, doing something, while at the same time we check the keyboard to see if the user wants to break into the program (to end the program, for example). BASIC programmers will recognize that this function operates like INKEY$.

Check Standard Input Status doesn't wait for a key to be pressed before going on to the next program instruction.

We check the keyboard status in lines 011D and 011F. Now, if the result returned in AL is FFh, we know that something has been typed and we want to exit from the program with an INT 21. Otherwise we want to return to put the magic number into the timer again, at "sounder." To find out what's in AL, we increment it. If it was FF, it will now be zero, and our JNZ instruction at location 0123 will "fall through" to the INT 21h. If it was 00 (nothing typed) it will be 1 when we increment it, and the JNZ will cause a jump back to "sounder."

Controlling Sound with the Keyboard

Our next programs are somewhat more ambitious. They are interactive programs, in that you can control, from the keyboard, the pitch of the tone being generated by the speaker.

The KAZOO Program

In this program you have four control keys:

1 — raises the pitch

2 — lowers the pitch

9 — turns the sound on

0 — turns the sound off

Once you press, say, "1", the tone will rise and continue to rise until you press either "2" to start it going down, or "0" to turn it off altogether. The resulting wailing is something like the sound of a kazoo, and lends itself to certain kinds of music, such as "The Flight of the Bumblebee."

There is a new pseudo-op in the program, EQU, which we'll talk about as soon as you've typed in the program and tried it out. Here's the listing:

```
                        ;KAZOO--Uses Timer2 to run speaker
                        ;  produces variable-pitch sounds

= ØØ61                  portB   equ     61h     ;I/O Port B
= ØØØ7                  keybd2  equ     7h      ;keybd input, no echo
= ØØØB                  status  equ     Øbh     ;check kbd status
= ØØ21                  doscall equ     21h     ;DOS interrupt number

                        ;*****************************************
Ø1ØØ                    prognam segment ;define code segment
```

```
                        ;-----------------------------------------
0100                    main    proc    far     ;main part of program

                                assume  cs:prognam

                                org     100h    ;start of program

0100                    start:                  ;starting execution address

                        ;initial values
0100  BB 0500                   mov     bx,500h ;set 1/pitch in BX
0103  B2 00                     mov     dl,0    ;set pitch change to 0
0105  B6 03                     mov     dh,3    ;set on/off status on

0107                    sounder:
0107  B0 B6                     mov     al,10110110b  ;put magic number
0109  E6 43                     out     43h,al        ;     into timer2
010B                    tone:
010B  8B C3                     mov     ax,bx   ;move 1/pitch into AX
010D  E6 42                     out     42h,al  ;LSB into timer2
010F  8A C4                     mov     al,ah   ;MSB to AL, then
0111  E6 42                     out     42h,al  ;  to timer2

0113  E4 61                     in      al,portB ;read port B into AL
0115  24 FC                     and     al,11111100b ;mask off bits 0, 1
0117  02 C6                     add     al,dh   ;add on/off status
0119  E6 61                     out     portB,al ;to turn speakr on/off

                        ;raise or lower pitch by amount in AX
011B  8A C7                     mov   al,bh     ;divide BX by 100h
011D  B4 00                     mov   ah,0      ;top half of AX = 0
011F  0D 0001                   or    ax,1      ;make sure at least 1
0122  0A D2                     or    dl,dl     ;does  DL = 0 ?
0124  74 02                     jz    skip      ;  if so, AX is plus
0126  F7 D8                     neg   ax        ;make AX negative
0128  03 D8             skip:  add    bx,ax     ;add change to pitch

012A  B9 0200                   mov   cx,200h   ;set up wait loop
012D  E2 FE             wait:  loop   wait      ;  loop a while


012F  B4 0B                     mov   ah,status ;check status function
0131  CD 21                     int   doscall   ;call DOS

0133  FE C0                     inc   al        ;if AL was FF, then
0135  74 02                     jz    read_key  ;  character was typed
0137  EB D2                     jmp   tone      ;sound tone again
```

```
                        ;read keyboard to get digit
                        ; 1=lower pitch, 2=raise pitch, 9=on, Ø=off

Ø139                    read_key:
Ø139  B4 Ø7                     mov  ah,keybd2  ;keybd funct, no echo
Ø13B  CD 21                     int  doscall    ;call DOS
Ø13D  3C 31                     cmp  al,'1'     ;is it 1 ?
Ø13F  74 ØE                     jz   lower      ;  lower pitch
Ø141  3C 32                     cmp  al,'2'     ;is it 2 ?
Ø143  74 ØE                     jz   higher     ;  raise pitch
Ø145  3C 39                     cmp  al,'9'     ;is it 9 ?
Ø147  74 ØE                     jz   turn_on    ;  turn on tone

Ø149  3C 3Ø                     cmp  al,'Ø'     ;is it Ø ?
Ø14B  74 ØE                     jz   turn_off   ;  turn off tone
Ø14D  EB BC                     jmp  tone       ;not recognized

Ø14F                    lower:
Ø14F  B2 ØØ                     mov  dl,Ø
Ø151  EB B8                     jmp  tone
Ø153                    higher:
Ø153  B2 Ø1                     mov  dl,+1
Ø155  EB B4                     jmp  tone
Ø157                    turn_on:
Ø157  B6 Ø3                     mov  dh,ØØØØØØ11b
Ø159  EB AC                     jmp  sounder
Ø15B                    turn_off:
Ø15B  B6 ØØ                     mov  dh,Ø
Ø15D  EB A8                     jmp  sounder


Ø15F                    main    endp    ;end of main part of program
                         ;-----------------------------------------

Ø15F                    prognam ends    ;end of code segment
                        ;*****************************************

                                end     start   ;end assembly
```

The EQU Pseudo-Op

When we use the instruction INT 21h we are doing an "interrupt number 21" call to DOS so that it can perform one of the DOS functions. The number 21h, used like this, is not particularly revealing of the purpose of the instruction. Anyone looking at the program listing who didn't know what the 21h meant would not understand the purpose of the INT instruction. It would be nice if there were a way to use a symbolic *name* for these numbers in the operand field, in the same way

that we use *labels* to stand for particular memory addresses.

This is the purpose of the EQU instruction. Look at the EQU statements at the top of the program:

```
portB   equ     61h     ;I/O Port B
keybd2  equ     7h      ;keybd input, no echo
status  equ     Øbh     ;check kbd status
doscall equ     21h     ;DOS interrupt number
```

EQU stands for "EQUivalent." The first program line in this section tells the assembler, "From now on, whenever you see the word 'portB', translate it into the number 61h." Similarly, "keybd2" is EQUivalent to 7h, "status" is 0Bh, and "doscall" is 21h.

There is no change in the output from the assembler when you use EQU statements. The instruction

```
int 21h
```

assembles exactly the same as

```
doscall equ 21h
        ...
        ...
        int doscall
```

The only difference is the way the ASM and LST files look. When adding an EQU pseudo-op will make your program easier to read and understand, then it's a good idea to use it. Otherwise, it's a complete waste of time.

When to use EQUs and when not to is largely a matter of style. Some programmers use lots of them, some not so many. Our philosophy is generally to use them when they stand for something external to the program like the numbers of DOS functions, but not to use them for numbers that are important in themselves, like masks and fixed numerical constants. But you can do whatever you want; no one will revoke your programming license.

We've used a few EQUs in the program above to ease you gradually into their use; later on we'll use more of them.

Operating KAZOO

The flow chart in Figure 7-5 shows the general structure of the KAZOO program.

There are two main parts to the program. The first, from the beginning to location 012D, the end of the "wait" loop, is involved with

creating the tone. The second part, from 012F to the end, reads what you type on the keyboard and makes appropriate changes. These two parts of the program communicate through the DH and DL registers.

The DH register holds a number which determines whether the tone will be on or off. The number 3 turns it on, while 0 turns it off. These

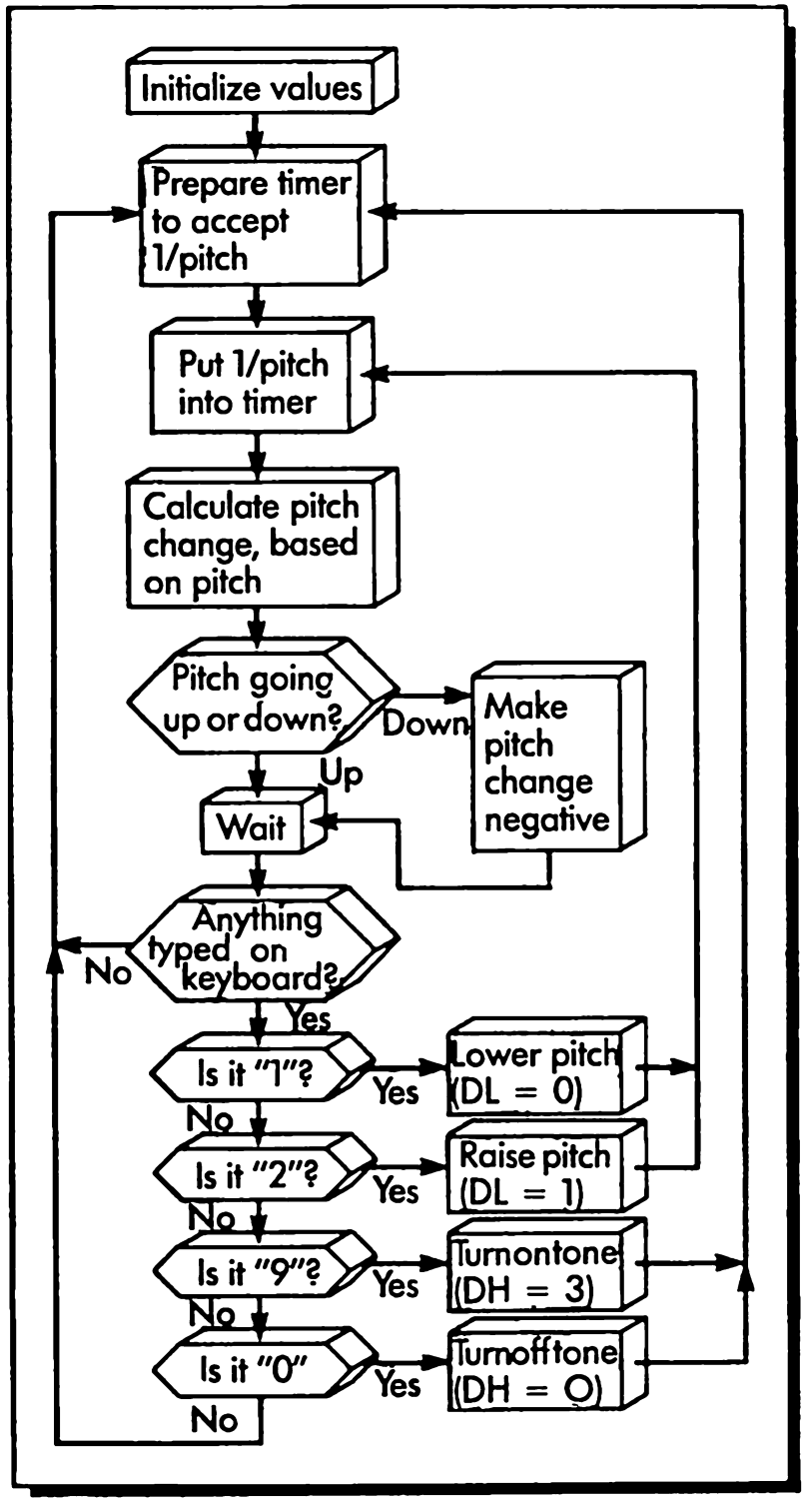


Figure 7-5. Flow chart of the KAZOO program

numbers are set up in the second part of the program in response to the user typing "9" or "0" on the keyboard. They are then, in the first part of the program, added to the number found in portB, which is port 61h (as defined in the EQU statement). The result is sent back to portB to turn the sound on or off, just as it has been in all the preceding sound programs.

The DL register holds either 0 or 1, depending on whether the pitch is rising or falling. If the pitch is rising, the number is 1, if falling, it's 0. Again, this number is set in the second part of the program, and used in the first to change the pitch.

When KAZOO is running normally, with nothing being typed on the keyboard, it cycles repeatedly through the top half of the program, changing the pitch each time, much like the SIREN program. The differences are that the pitch change is more sophisticated, being proportional to 1/pitch (as we'll see), and being either positive or negative (depending on DL); and that the tone can be either on or off (depending on DH). When something is typed on the keyboard, then the second half of the program comes into play, and the value in DH or DL is changed according to the character typed.

Changing the Pitch

Changing the pitch when the appropriate key is pressed is not as simple as it might seem. The problem is that the musical scale, at least as we perceive it, is arranged in a geometric progression rather than an arithmetic progression. That is, the higher the frequency of the note, the more you need to change the frequency to get to the next note. In other words, the numbers you must add to *low* frequencies to change them are smaller than the numbers you must add to *high* frequencies to change them the same amount. Another way to see this is to realize that to go from a particular frequency or musical note to one octave above that note, you don't add a fixed amount to the frequency, you *double* the frequency, whatever it is.

Although the number we're dealing with, "1/pitch," is the *inverse* of the frequency, the same principle applies. When this number is small, like 100, we want to be adding small numbers to it, like 1, to change the pitch. When it is large, like 20000, we want to add larger numbers to it, like 20.

To figure out how much to change 1/pitch (which is in the BX register), we'll divide it by 100h. This turns out to be easy to do: we simply take the high half of BX and put it in the low half of another register, AX. Then we make sure the high half of AX is zero. AX now

contains a number 1/100h the size of the one in BX. We add this number to BX, and we've changed 1/pitch by an amount proportional to 1/pitch, which is just what we set out to do. That's what the MOV AL,BH instruction at 011B and the ADD BX,AX at 0128 accomplish.

We don't want an increment of zero, or the sound would get stuck on the same note forever, so we OR on a 1 to AX, in location 011F. And we need to know whether to add or subtract the change in pitch, based on the number 0 or 1, in the DL register. We want to use the same ADD instruction even if we're going to subtract, and we do this by making AX negative with the NEG instruction at location 0126.

The NEG Instruction

NEG Instruction

Changes sign of operand.

The arithmetic sign of the operand is changed, using two's complement arithmetic.

```
NEG AL
NEG BX
```

Flags affected: AF, CF, OF, PF, SF, ZF

Thus if the AX register contains 10h, the instruction NEG AX will change it to –10h, which is FFF0h. 3h would be changed to -3h, which is FFFDh, and so on.

The actual process used by this instruction to change the sign is interesting for those of you who are two's complement arithmetic buffs. The instruction first *complements* all the bits in the original number. ("Complements" simply means to change 0 to 1, and 1 to 0.) NEG then adds 1 to the result. The result is the negative of the number.

```
0010h = 00000000 00010000 b     <— Start with this

        11111111 11101111 b     <— Complement each bit
               +        1       <— Add one
        -------------------
FFF0h = 11111111 11110000 b     <— Result is two's complement
```

The PIANO Program

Our next program is somewhat similar to KAZOO, in that it plays different notes and is controlled from the keyboard. However, PIANO, as the name implies, is much more structured in that it has a fixed repertoire of notes, each of which is sounded by pressing a particular key. The number keys (the ones on the top row, not the numeric keypad) are used to play the notes of the scale, with "1" being C, "2" being D, and so on up to "8", an octave above the first C. It's easy to play simple tunes with this program, but they must fall in that one octave. This is not a lot of notes, but if you want to join a rock group you will be able to make up in the uniqueness of your instrument what it lacks in range.

Actually, it's not hard to do what PIANO does by using BASIC, which has a nice set of statements for generating notes. However, our assembly-language program will introduce you to some important new instructions, and to the use of data stored in memory. Examining how the computer deals with data in memory will demonstrate that we still have some things to learn about the 8088, and will lead to some questions about memory segments, which we'll answer in the next chapter.

In fact, the somewhat convoluted way we have to go about writing this program so that it will work without segments, and so that it will convert to a COM file, will be considerable incentive to find out more about segments.

Here's the PIANO program:

```
                        ;PIANO--Uses Timer2 to run speaker
                        ;  number keys play notes of the scale

= ØØ61                  portB   equ     61h     ;I/O Port B
= ØØØ7                  keybd2  equ     7h      ;keybd input, no echo
= ØØ21                  doscall equ     21h     ;DOS interrupt number
= ØØØ3                  cont_c  equ     Ø3h     ;control-C ASCII code

                        ;*****************************************

ØØØØ                    prognam segment ;define code segment

                        ;-----------------------------------------
ØØØØ                    main    proc    far     ;main part of program

                                assume  cs:prognam

Ø1ØØ                            org     1ØØh    ;first address

Ø1ØØ                    start:          ;starting execution address
```

```
                        ;read keyboard to get digit from Ø to 7
Ø1ØØ                    read_key:
Ø1ØØ  B4 Ø7                     mov   ah,keybd2 ;keybd funct, no echo
Ø1Ø2  CD 21                     int   doscall   ;call DOS
Ø1Ø4  3C Ø3                     cmp   al,cont_c ;is it control-C ?
Ø1Ø6  74 34                     jz    exit      ;yes, so exit
Ø1Ø8  2C 31                     sub   al,31h    ;change ASCII to digit
Ø1ØA  24 Ø7                     and al,ØØØØØ111b ;mask off uppr 5 bits
Ø1ØC  DØ EØ                     shl   al,1      ;* by 2 (2 bytes/word)
Ø1ØE  98                        cbw             ;byte --> word in AX
Ø1ØF  8B D8                     mov   bx,ax     ;put in BX (for table)
Ø111  B8 ØØØØ                   mov  ax,Ø       ;numerator (low word)
Ø114  BA ØØ12                   mov  dx,12h     ; (high word)
Ø117  F7 B7 Ø13E                div  [13Eh + bx] ;divisor from table
Ø11B  8B D8                     mov   bx,ax     ;save quotient in BX

                        ;set 1/pitch into timer, then turn on tone
Ø11D  BØ B6                     mov   al,1Ø11Ø11Øb  ;put magic number
Ø11F  E6 43                     out   43h,al    ;     into timer2
Ø121  8B C3                     mov   ax,bx     ;1/pitch into AX
Ø123  E6 42                     out   42h,al    ;LSB into timer2
Ø125  8A C4                     mov   al,ah     ;MSB to AL, then
Ø127  E6 42                     out   42h,al    ;  to timer2
Ø129  E4 61                     in    al,portB  ;read port B into AL
Ø12B  ØC Ø3                     or    al,3      ;turn on bits Ø and 1
Ø12D  E6 61                     out   portB,al  ;to turn on speaker


                        ;sound note for a while, then turn it off
Ø12F  B9 FFFF                   mov   cx,ØffffH ;set up for delay
Ø132  E2 FE             wait:   loop  wait      ;delay
Ø134  E4 61                     in    al,portB  ;read port B into AL
Ø136  24 FC                     and   al,111111ØØb ;mask lower 2 bits
Ø138  E6 61                     out   portB,al  ;to turn off speaker
Ø13A  EB C4                     jmp   read_key  ;go get another digit


                        ;control-C typed, so exit
Ø13C                    exit:
Ø13C  CD 2Ø                     int   2Øh       ;return to DOS

                        ;frequencies of notes
Ø13E  Ø1Ø6                      dw      262d    ;C
Ø14Ø  Ø126                      dw      294d    ;D
Ø142  Ø14A                      dw      33Ød    ;E
Ø144  Ø15B                      dw      347d    ;F
Ø146  Ø188                      dw      392d    ;G
```

```
Ø148  Ø1B8                    dw      44Ød    ;A
Ø14A  Ø1EE                    dw      494d    ;B
Ø14C  Ø2ØC                    dw      524d    ;C

Ø14E                    main    endp    ;end of main part of program
                        ;-----------------------------------------
Ø14E                    prognam ends    ;end of code segment
                        ;*****************************************
                        ;
                                end     start   ;end assembly
```

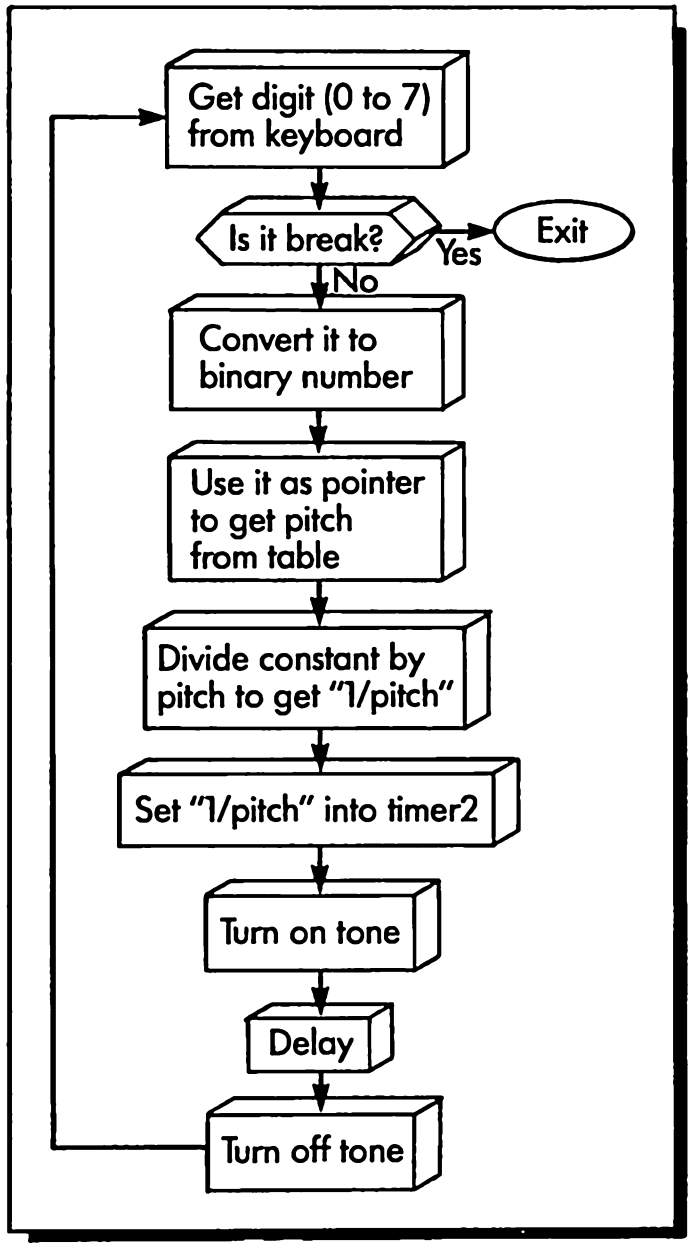


Figure 7-6. Flow chart of the PIANO program

Figure 7-6 shows a flow chart of the program. As you can see it is quite straightforward, and in many ways similar to the previous sound program, KAZOO. The interesting part is how we find the frequency of the note we want to play. The frequencies of the notes are arranged in a table at the end of the program. If we type a "1", we want to get the first one of these frequencies from the table, and divide it into a fixed constant — which turns out to be 120000h — to get the "1/pitch" number which is to be put into the timer. (Of course, we don't really mean "1/pitch," we mean "120000/pitch," but that's hard to write.) If we type "2" we want the second frequency, and so on. This process is shown in Figure 7-7.

To get the appropriate frequency from the table we want to change the digit which was typed in, into a *pointer* to the table of frequencies. We'll put this pointer in the BX register, and then, using indirect addressing, we'll be able to access the appropriate frequency. To do this we first turn the digit which was typed on the keyboard into binary — and subtract 1 at the same time — by subtracting 31h. We make sure the result is in the range 0 to 7 by ANDing off the upper 5 bits. Then we multiply by two, since there are two bytes for every frequency in the table. Multiplying by two is always easy in assembly language: all you have to do is shift your number one bit to the left. To do this, we use a new instruction, SHL.

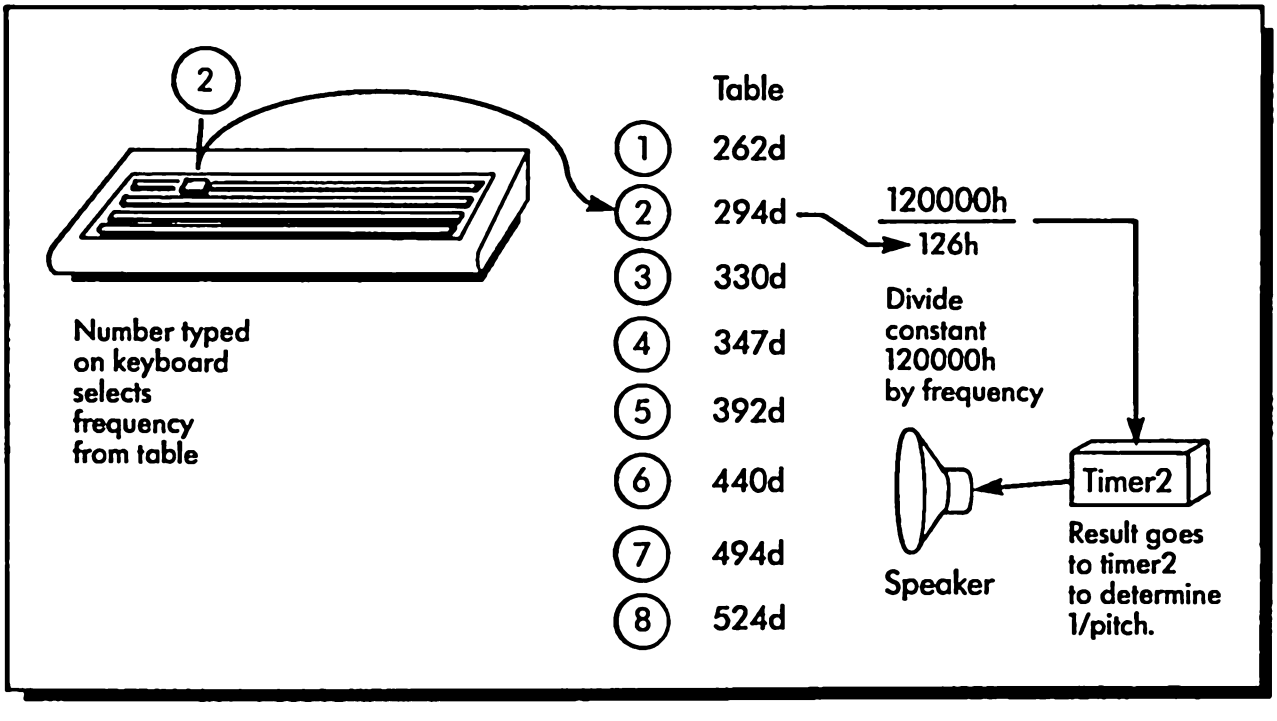


Figure 7-7. Operation of the PIANO program

The SHL Instruction

SHL Instruction

Shifts a register Left. (SHL stands for "SHift logical Left." Also called SAL, for "Shift Arithmetic Left.")

All bits in register move left.

Bits from left-hand end appear in the carry flag, 0 bits are shifted into the right-hand end.

To shift 1 bit:

```
SHL DL,1
```

To shift more than one bit, put the number in CL first:

```
MOV CL,3
SHL BX,CL
```

Flags affected: CF, OF, PF, SF, 2F

The difference between shift instructions and rotate instructions is that in *shifts* the bits are pushed off one end of the register and zeros are shifted in at the other end. In *rotates* the bits from one end rotate around to the other end. Bits aren't lost in rotates, but in shifts the bits shifted out of the end of the register are gone forever (except that the last one shifted is in the carry flag).

A shift is more appropriate than a rotate for multiplying by 2, since we don't want any bits that might have been left over in the high bit positions to rotate around and get into the low-order positions. (Yes, we know, we masked them off, but it's the principle of the thing.)

Using Indirect Addressing with the Assembler

Once we get our pointer (the original digit we typed, converted to binary, minus 1, times 2), we want to put it in BX so we can use indirect addressing to get the frequency out of the table. (Reread the section on indirect addressing in chapter 4, if you've forgotten what it is.) We're going to take this frequency, when we get it, and divide it into a constant to arrive at 1/pitch. (If you want you can "tune" your IBM piano — move all the notes up or down at once — by changing this constant.) Let's see how we do division with the 8088.

DIV Instruction

DIVides two numbers.

Divides contents of A register by contents of operand register (or memory address).

To divide a word by a byte, put word in AX, byte in operand register (or memory location).

```
MOV AX,NUMERATOR
DIV DIVISOR (8-bit register or addr)
Quotient will be in AL
Remainder will be in AH
```

To divide a double-word by a word, put double-word in DX and AX, and word in operand register (or memory address).

```
MOV DX,HI_NUMERATOR
MOV AX,LO_NUMERATOR
DIV DIVISOR (16-bit register or addr)
Quotient will be in AX
Remainder will be in DX
```

Flags affected: flags are undefined after DIV

If you want to divide a byte by a byte, you have to first make the numerator in AL into a word in AX by using the CBW (convert byte to word) instruction, and if you want to divide a word by a word, you have to first make the numerator into a double-word in the DX+AX double-register, using the CWD (convert word to double-word) instruction. A double-word is simply a 32-bit quantity occupying two registers.

We need to use double-word division because we want our result to have the accuracy of a word, not a byte — we can't "fine-tune" the 1/pitch number enough if it's only eight bits long. To create the double-word constant we place the number 120000h into DX and AX in two separate instructions.

Now we come to the $64,000 dollar question: what do we divide by? Well, we divide by the 16-bit quantity we find in the table of frequencies at a certain address. This address is found by adding the pointer, which we derived from the typed-in digit above, to the address of the start of

the table — 013Eh. Since we've placed our pointer in BX, we can express this address as "13Eh + bx", where bx really means the *contents of the BX register.*

Thus if BX contained four, the "effective address" would be 13Eh + 4, or 142h. The square brackets mean "indirect addressing," so (assuming BX still contains 4), [13eh + bx] means the *contents of memory location 0142h.* If we had typed "3" (whose ASCII code is 33h), it would have been converted into 4 (by subtracting 31 and multiplying by 2). This corresponds to the third entry in the table, which is at address 0142h... just what we want. Now we have both the numerator and the divisor for our division, which looks like this:

$$1/\text{pitch} = \frac{1200000\text{h}}{\text{frequency from table}}$$

The result of the division is what we call 1/pitch, the number we send to timer2 to specify the tone. In the next part of the program we simply turn on this tone, wait a while, turn it off again, and go back to read another note at the keyboard. Unlike the KAZOO program, we *wait* for a digit to be typed before sounding a tone.

You'll find that the keyboard has a rather strange "touch" for a piano, and that the keyboard buffer lets you type faster than the program can play the notes. If you want a surprise, try typing on the function keys, F1 through F10.

Type in this program and try it out! Who knows, maybe you'll discover, buried deep within yourself, a hidden talent for music. You've already discovered such a talent buried deep within the computer.

Tuning Up the PIANO

Besides certain musical limitations, PIANO has a glaring defect from a programmer's viewpoint. Have you seen what it is? It's the fact that we had to use *an absolute number* to specify the address of the operand in the DIV instruction in line 0117. This absolute address is 013Eh.

```
0117  F7 B7 013E          div  [13Eh + bx] ;divisor from table
                                  |
                                  how did we know what number to put here?
```

Have you asked yourself how we knew what number to put in the brackets when we were writing the program? The fact is, we didn't. Since we didn't yet have a LST file, we didn't know where the frequencies were going to come out. We had to assemble the program *twice*, once to find

out this address, and the second time for real. Obviously we don't want to use a programming technique that requires us to do our assemblies twice. There could hardly be anything more inelegant, not to mention time-consuming.

What we really wanted to do, of course, was to use a label for the table (called, for instance, "table"):

```
                        ;frequencies of notes
                        table
013E  01Ø6                    dw      262d    ;C
014Ø  0126                    dw      294d    ;D
0142  014A                    dw      33Ød    ;E
0144  015B                    dw      347d    ;F
0146  0188                    dw      392d    ;G
0148  01B8                    dw      44Ød    ;A
014A  01EE                    dw      494d    ;B
014C  02ØC                    dw      524d    ;C
```

Then we could have referred to this symbolic address instead of an absolute numerical address, and let the assembler figure out where "table" was:

```
0117  F7 B7 013E          div  [table + bx] ;divisor from table
```

But this doesn't work! Why not? On the answer to this question hangs a long and complex tale.

How the Assembler Thinks about Segments

If we refer to a label in a jump or call instruction, there's no problem. However, if we refer to a label in an instruction which is concerned with data, such as MOV, or an arithmetic or logical instruction like ADD, we'll have trouble. If the data (with its label) *follows* the instructions that refer to it, the program won't assemble properly. If the data *precedes* the instruction that refers to it, the EXE file won't convert to a COM file.

One reason for all this difficulty is that *the assembler assumes that data are kept in a different memory segment from the program.* Another is that we are using COM files.

In the next chapter we're going to explain memory segments and EXE files in detail, and then we'll learn the answer to the problems posed above.

Summary

In this chapter you've practiced your assembly-language skills on the sound-producing facilities of the PC. You now know how to make different kinds of sounds using two basic sound-producing methods: toggling a gate on and off, and sending a 1/pitch number to timer2.

You've also learned about the stack, an extremely important and versitile feature of the 8088. We'll be coming back to the stack in later chapters. You've learned some new 8088 instructions, and about EQU, the pseudo-op that makes constants easier to understand. And finally, confronted with some problems of storing data in our program, you've learned that we still have a lot left to learn about memory segmentation and EXE files.

8

Memory Segmentation and EXE Files

Concepts

Memory segmentation
Segment registers
EXE files
SI and DI registers
String-handling instructions

8088 Instructions

REP = Repeat
MOVS = Move string
CMPS = Compare strings
SCAS = Scan string
CLD = Clear direction flag
STD = Set direction flag
REPE = Repeat while equal (or REPZ = Repeat while zero)
REPNE = Repeat while not equal (or REPNZ = Repeat while not zero)

There are two kinds of files which can be executed by the 8088 microprocessor: COM files and EXE files. Thus far in this book we've shown executable programs in the form of COM files. The advantages of this approach are that COM files are simpler to understand, and require less overhead (in the form of a minimum number of statements needed to make a program run) than EXE files do. COM files are also faster to load and take up less memory space than EXE files, so they have a definite place in your PC's system for relatively small programs that don't do a lot of data manipulation.

However, for larger programs, EXE files have many advantages. One is that a number of EXE files can be combined, using LINK, to produce a single executable program. (We'll use this technique later in the chapter on interfacing with higher-level languages.)

More important, however, is the fact that EXE files can make full use of the *memory segmentation* feature of the 8088 microprocessor. Thus you can put your program in one 64K segment, your data in a second segment, your stack in a third, and additional data in a fourth — thus utilizing up to four times 64K or 256K of memory for a single program. (And with a small amount of additional effort you can even go beyond these limits.) This means not only that your programs can be larger, but that they can be more clearly structured. Modern progamming techniques favor dividing programs into distinct parts whenever it will improve clarity. In addition, several EXE routines, combined into a larger program, can also share common data and stack segments.

EXE files are also the more "standard" format for an executable program on the IBM PC: they are what IBM expects you to use most of the time. The MACRO Assembler, the linker, and the operating system are all set up to produce and execute EXE files with the minimum amount of fuss.

One purpose of this chapter is to introduce you to the use of EXE files. However, in order for you to understand EXE files, you must also understand memory segmentation, which is itself an essential topic for anyone writing programs on the PC. We will, therefore, discuss both memory segmentation and EXE files more or less simultaneously.

Later in the chapter we will also introduce the 8088's special string-handling instructions. These instructions, which operate on strings of bytes (or words) at a time, make use of memory segments in an interesting way. Studying how they work will therefore increase your understanding of the use of memory segments and the writing of EXE files.

Memory Segmentation

In chapter 1 we introduced the two-part addresses used in the 8088: addresses of the form 0915:0100, where 0915 is the *segment address*, and 0100 is the *offset address*. This two-part memory system is necessary because the 8088 uses *16-bit registers*, and at the same time must be able to address up to 1 megabyte of memory, which requires *20 bits*. Let's review how this works, so that we'll be better able to understand the use of the segment registers.

Two-Part Addressing

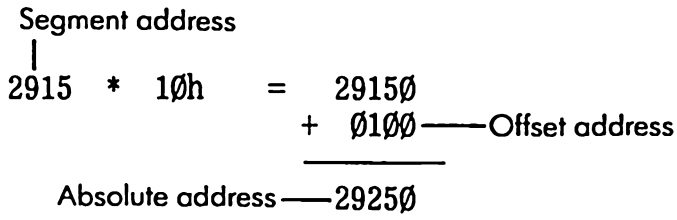
One megabyte is 1,048,576d bytes. The hexadecimal equivalent of this number is 100000h, so the hex addresses in a memory this large will run from 0h to FFFFFh. These addresses, which use a single number to specify each address in the one megabyte address space, are called "absolute addresses." Thus we require absolute addresses of *five* hex digits, or 20 bits, to address the entire memory in the 8088. However, the 8088 employs *16-bit architecture,* meaning that the internal registers are all 16 bits or *four* hex digits wide.

A logical question is, if it requires five hex digits to address the entire 8088 memory, why not simply use registers this size? The reason the 8088 uses 16-bit registers is that 16 bits is an even multiple of eight bits, or one *byte*. The byte has become the standard unit of information in the computer industry, largely because it is a convenient size for representing a single ASCII character. Registers which are not a multiple of a byte would create all sorts of compatibility problems when reading data from 8-bit peripherals, so in general the registers in all computers are multiples of one byte (eight bits): either 8, 16, 32, or (in the case of huge mainframe computers) 64 bits wide.

Why not use 16-bit registers to hold data bytes, and other 20-bit registers to hold memory addresses? Because having registers of two different sizes would cause a great increase in the complexity of the computer. What would happen when we tried to transfer a 16-bit quantity into a 20-bit register? Or vice versa? It would all be rather messy and complicated.

> Memory Segmentation is a way of making two 4-digit registers do the work of one 5-digit register.

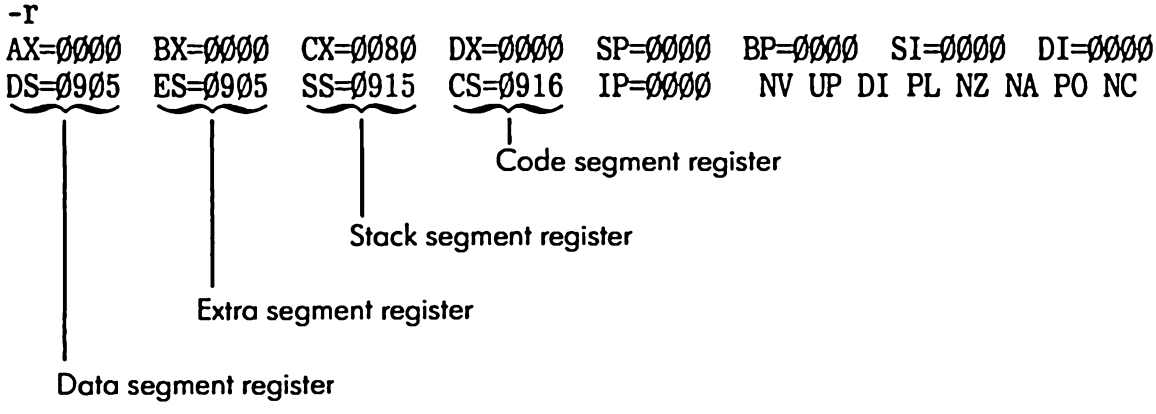
The designers of the 8088 thus came up with the scheme in which two 16-bit registers are used to generate one 20-bit memory address. The first of these registers contains part of the address, called the *segment address*, and the second contains another part of the address, called the *offset address*. As we learned, the absolute memory address is derived from the segment and offset addresses by multiplying the segment address by 10h, and then adding the offset address to the result. For example, the segment:offset address 2915:0100 is translated into its corresponding absolute address like this:



Segment Registers

The offset part of the address of a particular instruction being executed in a program is, as we've noted in previous chapters, contained in the IP register. Where is the corresponding segment address? It's in the *code segment register,* which is one of four *segment registers.* Thus if we know what's in the code segment register, and we know what's in the IP register, we can calculate the absolute address of a particular instruction. More to the point, the 8088 can calculate this address, so that it can figure out where to find the instruction.

One of the questions confronting the designers of the 8088 was how many segment registers to have. If you use the "R" command in DEBUG, you'll see the answer they came up with:



The number of registers turns out to be four: one segment register (CS) to hold the segment address of the program's instructions, one (SS) to hold the segment address of the stack, and two (DS and ES) to hold the segment addresses of two areas of data: the "data segment" and the "extra segment."

Why not three segment registers, or five or six? Clearly four was a compromise between having too few to do the job conveniently, and having so many that the design and operation of the 8088 became unnecessarily complicated. However, four is a reasonable number. There are three different kinds of things that need to be addressed: code (program instructions), the stack, and data. Each of these gets its own

register, except that data get two. One reason for having two data registers will become clear when we talk about string-handling instructions.

Segments

When you put a number in a segment register you have in effect defined something called a "segment." This is a section of memory 64K (65536d, or 10000h) bytes long (one sixteenth of the total 8088 address space.) If the segment address is, for example, 0916h, then the addresses in this segment start at 0916:0000 and go up to 0916:FFFF, which is the highest address in this particular segment. The relationship between a segment and the register which defines it is shown in Figure 8-1.

Since the segment address in effect always has a zero in the rightmost

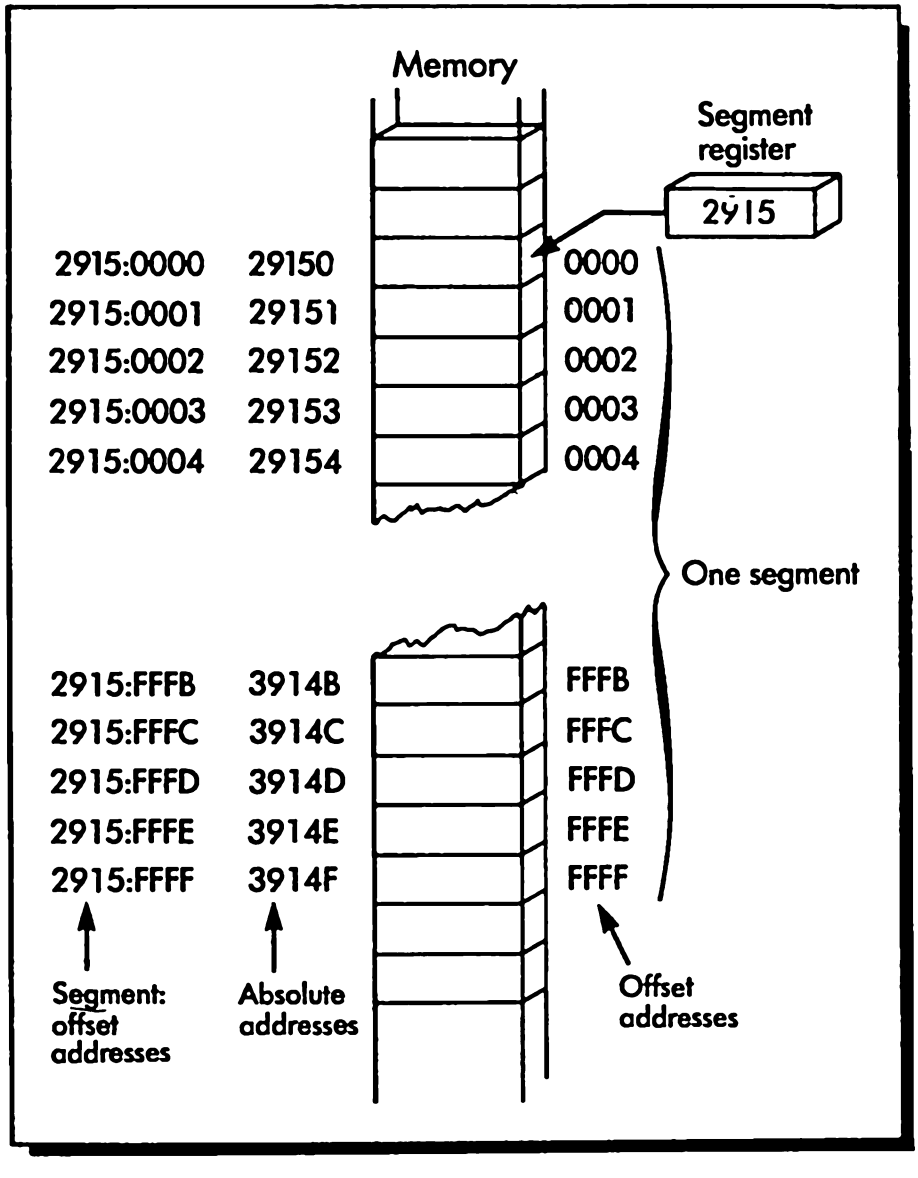


Figure 8-1. Segment and segment register

position (because it was multiplied by 10h), segments can only begin at addresses which are multiples of 10h (16d). In the 8088 groups of 10h (16d) bytes are called "paragraphs," so we can say that segments must start on a "paragraph boundary."

Where can you put the segments in memory? Just about anywhere you want. They can occupy completely separate parts of memory, they can overlap, and two or more segments can even occupy exactly the same space. Figure 8-2 shows some possible arrangements. As you'll see soon, a typical EXE file has an arrangement of its own.

In the next section we're going to look at our first EXE program. You'll find that many of the differences between EXE and COM files have to do with the segment registers, so let's plunge ahead and see what they are.

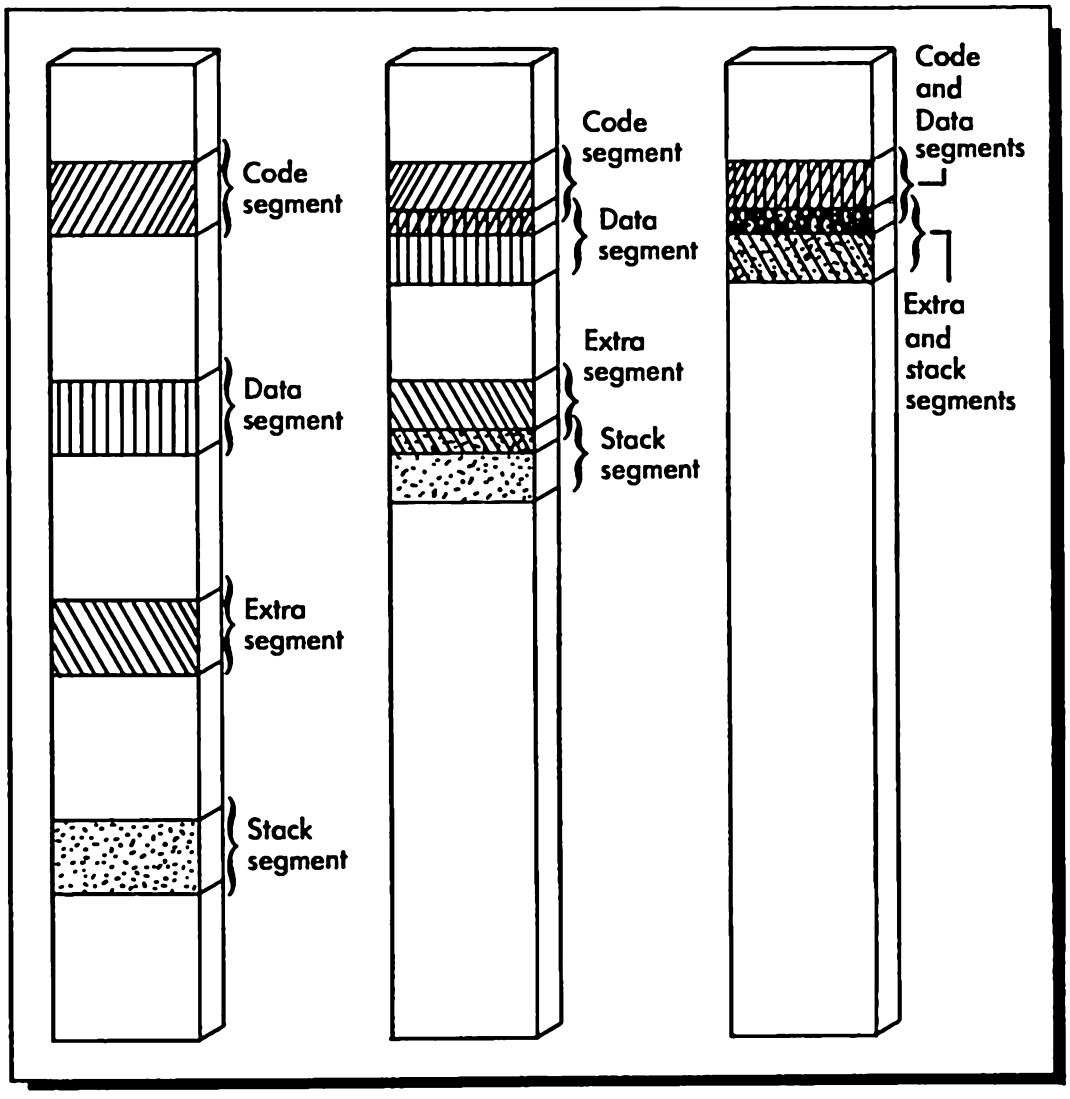


Figure 8-2. Possible arrangements of segments in memory

The PSTRING Program

In this section we're going to rewrite a program you last saw long ago in chapter 4. It was the program called PSTRING which printed a string on the printer, using indirect addressing with the BX register. The characters of the string were stored in memory using a "DB" pseudo-op. Then the program printed them one at a time. BX pointed to the location of each character in the buffer, and CX was used to hold a count of the number of characters, decremented each time one was printed so the program would know when to stop. Here's how the program looked when we typed it in using the "A" command:

```
Ø9Ø5:Ø1ØØ  mov  cx,ØØ31
Ø9Ø5:Ø1Ø3  mov  bx,Ø111
Ø9Ø5:Ø1Ø6  mov  dl [bx]
Ø9Ø5:Ø1Ø8  mov  ah,Ø5
Ø9Ø5:Ø1ØA  int  21
Ø9Ø5:Ø1ØC  inc  BX
Ø9Ø5:Ø1ØD  loop Ø1Ø6
Ø9Ø5:Ø1ØF  int  2Ø
Ø9Ø5:Ø111  db   'She is most fair, though she be marble-hearted.',Ød,Øa
```

And here's the corresponding listing using the "U" command:

```
Ø9Ø5:Ø1ØØ B931ØØ        MOV  CX,ØØ31
Ø9Ø5:Ø1Ø3 BB11Ø1        MOV  BX,Ø111
Ø9Ø5:Ø1Ø6 8A17          MOV  DL,[DX]
Ø9Ø5:Ø1Ø8 B4Ø5          MOV  AH,Ø5
Ø9Ø5:Ø1ØA CD21          INT  21
Ø9Ø5:Ø1ØC 43            INC  BX
Ø9Ø5:Ø1ØD E2F7          LOOP Ø1Ø6
Ø9Ø5:Ø1ØF CD2Ø          INT  2Ø
```

As we saw in the PIANO program in the last chapter, difficulties can arise when we try to use symbolic labels for data in COM files. For instance, you can't say "MOV BX,TABLE", where TABLE stands for a memory location. You have to say "MOV BX,131", or whatever the hex address of DATA is. In the simple approach developed in the last chapter, we had to assemble the program twice to discover what this number was.

Actually, if you know how to manipulate segments, it is possible to use data labels in COM files. One approach is to put the address of the data segment into the code segment (CS) register. However, this does not remedy any of the disadvantages of COM files discussed earlier in the chapter. The size of the program, including data and the stack, is still

restricted to 64K, programs cannot share data and stack segments, and cannot be linked together. We will therefore ignore this approach.

The easiest and most natural way to deal with data in a program is to put the data in a *separate segment*, and to do this we must use an EXE file. Let's see how to write PSTRING as an EXE file.

```
                                ;PSTRING--Program to print a string
                                ;  Demonstrates EXE files

= ØØØ2                          display equ     2h  ;display output function
= ØØ21                          doscall equ     21h ;DOS interrupt number

= ØØ31                          count   equ     49d ;# of characters in string

                                ;*******************************************

ØØØØ                            datarea segment         ;define data segment

                                ;string to be printed

ØØØØ  53 68 65 2Ø 69 73        string  db 'She is most fair, though she be '
      2Ø 6D 6F 73 74 2Ø
      66 61 69 72 2C 2Ø
      74 68 6F 75 67 68
      2Ø 73 68 65 2Ø 62
      65 2Ø
ØØ2Ø  6D 61 72 62 6C 65                db 'marble-hearted.',Ødh,Øah
      2D 68 65 61 72 74
      65 64 2E ØD ØA

ØØ31                            datarea ends
                                ;*******************************************

ØØØØ                            prognam segment         ;define code segment

                                ;-------------------------------------------
ØØØØ                            main    proc    far     ;main part of program

                                        assume  cs:prognam,ds:datarea

ØØØØ                            start:          ;starting execution address

                                ;set up stack for return
ØØØØ  1E                               push ds          ;save old data segment
ØØØ1  2B CØ                            sub  ax,ax       ;put zero in AX
ØØØ3  5Ø                               push ax          ;save it on stack
```

```
                                ;set DS register to current data segment
0004  B8  ---- R                       mov  ax,datarea ;datarea segment addr
0007  8E D8                            mov  ds,ax      ;  into DS register

                                ;PRINT CHARACTERS FROM STRING ON SCREEN

0009  B9 0031                          mov  cx,count   ;put # of chars in CX
000C  BB 0000 R                        mov  bx,offset string ;addr of string

000F                            next_char:
000F  8A 17                            mov  dl,[bx]    ;put one char in DL
0011  B4 02                            mov  ah,display ;Display Char function
0013  CD 21                            int  doscall    ;call DOS
0015  43                               inc  bx         ;advance pointer
0016  E2 F7                               loop next_char  ;repeat until done

0018  CB                               ret             ;return to DOS

0019                            main    endp    ;end of main part of program
                                ;-----------------------------------------
0019                            prognam ends    ;end of code segment
                                ;*****************************************
                                ;
                                       end     start   ;end assembly
```

Differences between COM and EXE Files

There are a lot of unfamiliar features in the program shown above. We'll go through these items one by one. When we're done, you'll know how to write an EXE file (or more accurately, how to write an ASM file which can be successfully assembled and turned into an EXE file with the linker).

Data and Code Segments

The first thing to notice in our program is that it has *two* segments now, rather than the single segment we have seen thus far with COM files. This new segment is the *data* segment, and is specifically for data that your program is going to operate on. In general form it looks like this:

```
                                ;*********************************************

0000                            datarea segment         ;define data segment

                                ;DATA GOES HERE
```

```
ØØ31                            datarea ends            ;end code segment
                                ;*******************************************
```

In PSTRING the data consists of a string defined by two "DB" statements.

The program instructions go in the *code* segment. In general form the code segment looks like this:

```
                                ;*******************************************

ØØØØ                            prognam segment         ;define code segment

                                ;PROGRAM GOES HERE

ØØ19                            prognam ends            ;end of code segment
                                ;*******************************************
```

As discussed before, we use asterisks in comment lines to separate one segment from another, for clarity in the listing. Both segments start with a *segment* pseudo-op, and end with an *ends* pseudo-op.

The ASSUME Pseudo-Op

ASSUME tells the MACRO Assembler which segments in your program listing will be associated with which segment registers. You have already used the ASSUME pseudo-op for COM files, in the form "assume cs:prognam." In general, we need an ASSUME statement for every segment, so we need to add one to cover our new data segment. Two or more ASSUMEs can be combined on the same line simply by separating them with commas, as we've done in the program above:

```
assume  cs:prognam,ds:datarea
```

We could also have used two separate statements and accomplished the same thing:

```
assume      cs:prognam
assume      ds:datarea
```

As you recall, there is a subtle point about ASSUME: it is (as are all pseudo-ops) an instruction *to the assembler program*, not an instruction for the 8088 microprocessor, or to your program. This is important to keep firmly in mind, as we'll see when we talk about setting up the DS register.

Setting Up the Stack for Return

We're going to use a different procedure for returning from EXE files to DOS (or DEBUG) than we did with COM files. Before, we used the INT 20h ROM call. It is possible to use this same system for returns from EXE files, but it is more usual to use the new system we're about to show you. It's the standard one recommended by IBM.

The new return procedure simply uses a RET instruction to transfer control from your program back to DOS or DEBUG. This causes one additional complexity. When we used the "INT 20" instruction before, the operating system took care of making sure the stack was restored to its proper value after the program was over. With the RET instruction, however, the responsibility of restoring the stack falls on the writer of the EXE program (that's *you*), rather than on the operating system.

So when we first start our program we must initialize the stack so that it's ready for our return. To do this we need to put two things on the stack: first, the original value in the DS register (since, as we'll see, our program is going to change the value in this register); and second, a zero. Here's how the code looks:

```
                                ;set up stack for return
0000  1E                        push ds          ;save old data segment
0001  2B C0                     sub  ax,ax       ;put zero in AX
0003  50                        push ax          ;save it on stack
```

We first PUSH the old value of DS onto the stack. The operating system needs the value of DS because it needs to restore this value when it takes control back from our program; otherwise it won't know where its data is. (Just as our program won't know where *its* data is unless we set the DS register correctly). Our technique of subtracting AX from itself is simply a quick way to generate a zero; then we PUSH the zero onto the stack.

> Telling the assembler what segment goes where, and telling the 8088, are two different things.

Setting Up the DS Register

We've already noted that we use the ASSUME pseudo-op to tell the MACRO Assembler which segment register it can expect to be used for which segment of the program. However, telling the MACRO Assembler

what's happening is only half the story. We must also *actually put the starting address of each segment into the appropriate segment register.* This is not true of the code segment, which is always set up correctly by the operating system. It is, however, true of the other three segments. For the time being we're ignoring the stack segment and the extra segment, so we don't need to worry about them; but we definitely do need to worry about the data segment; we need to put the address of the data segment in the data segment register. The code to do that looks like this:

```
                              ;set DS register to current data segment
0004  B8  ---- R                    mov  ax,datarea ;datarea segment addr
0007  8E D8                         mov  ds,ax      ;  into DS register
```

The dashed line following the B8 in location 0004 indicates that the assembler program did not know the address of the data segment DATAREA. We'll have more to say about this later.

As it turns out, there is no 8088 instruction which will load an address into a segment register directly. You must first MOVe the address into another register, in this case AX, and then MOVe it into the segment register.

If you forget this step — putting the address of the data segment in the DS register — the program will assemble correctly since the assembler doesn't know or care what is actually in any of the registers. It only knows what you've told it to assume is in the registers with the ASSUME statement. But when you *execute* the program, strange (generally bad) things will occur because the program — as a result of having the wrong address in the DS register — will be looking in the wrong part of memory, that is, in the wrong segment, for its data.

Origin at 0000

COM files always start at offset address 0100h in their particular segment, but EXE files always start at 0. There is therefore no need for an ORG statement at the beginning of EXE files, since the default origin with no ORG statement is 0.

Changing INT 20 to RET

As we've mentioned, we're going to use a plain old RET instruction in our EXE files, rather than the INT 20 we've used so far. *This must always be a FAR RET.* It must therefore be used only in a FAR PROCedure. Thus our program starts with

```
main proc  far
```

and ends with

```
main endp
```

"Main" is simply the name we give to the main part of the program; it could have been any name. (Re-read the sections on CALL, PROC, and RET in chapter 6 if you don't remember the differences between NEAR and FAR PROCedures.)

> Code labels are followed by a semicolon. Data labels are not.

Using Labels with Data

Notice the way we use a label to identify a particular data item:

```
string  db 'She is most fair, though she be '
  |
Label
```

In the past when we've used labels in the code segment they have been followed by a colon, as in "start:" and so on. We can use labels in the same way to identify data items, but they are *not* followed by a colon. If you forget and put the colon there, you'll get error messages. This is another way in which the MACRO Assembler tries to save you from yourself, by keeping you from getting confused between labels for data and labels for program locations.

What about the Extra Segment and the Stack Segment?

There is no Extra Segment in this program, so you don't need to worry about it at all. (We'll use it later in the chapter, so you'll see how it works then.) However, there *is* a stack; yet there is no Stack Segment (SS) in the program listing, and the program doesn't try to load anything into the SS register. How is it that we can simply ignore the SS register this way?

We're just lucky. When DOS (or DEBUG) turns control of the computer over to our program, the SS register is set to 0000. Also, the SS register is set to the same value as the data segment. Now, as you've learned already, the stack grows *downward* in memory from the *top* of its segment. If the SP (Stack Pointer) register is set to 0000, the first item placed on the stack will go at FFFE and FFFF, and subsequent values will be placed at addresses below that.

Thus the program and the data grow upward in memory; and the stack, which starts out about 64K bytes above the program, grows downward. If our program is short, and if the area occupied by the data is small, and if the stack doesn't get too big, the program and the stack will not run into each other. Figure 8-3 shows how the stack, the data, and the program segments relate to each other, for the case of our PSTRING program being loaded with DEBUG.

Would you like to see the actual values assigned to the segment

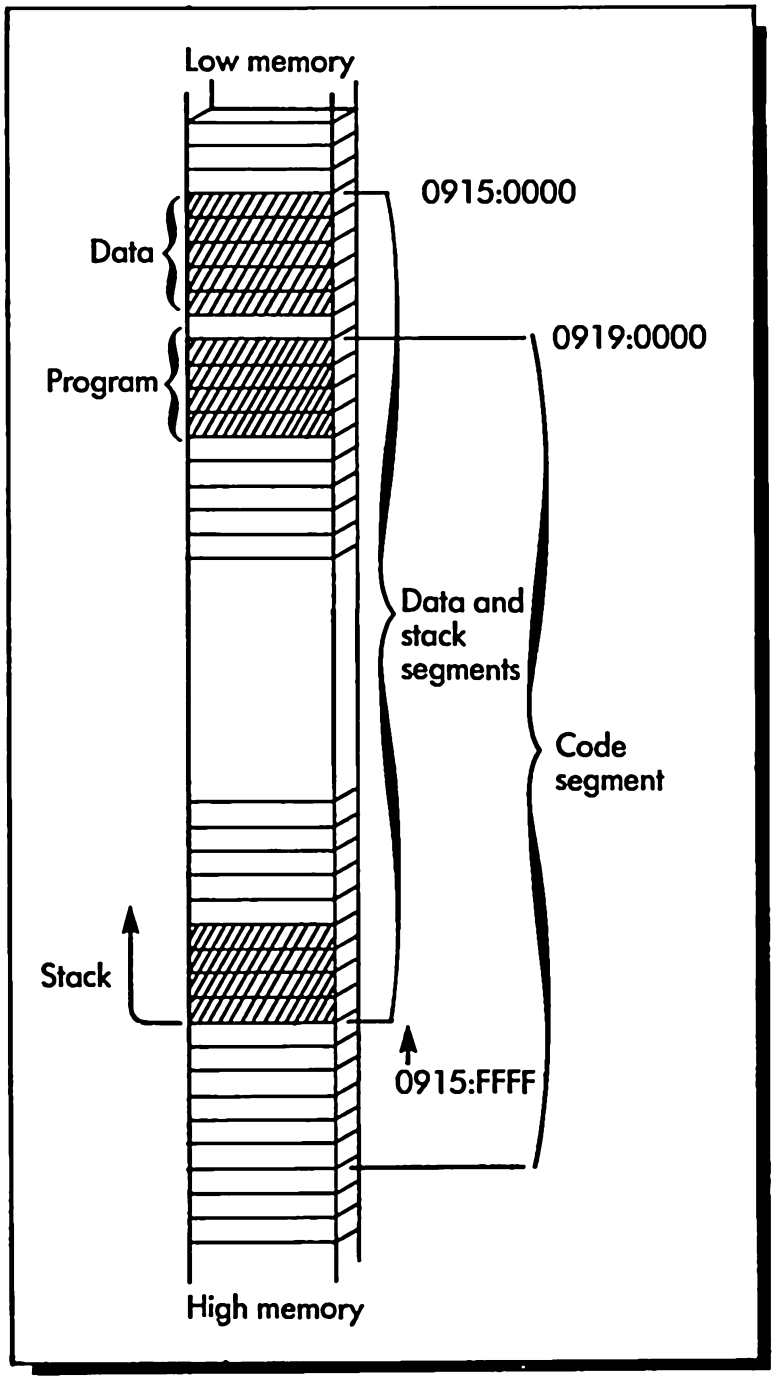


Figure 8-3. Stack, code, and data segments in a small EXE file

registers? Simplicity itself. Just load DEBUG and PSTRING.EXE and look at the registers with the "R" command:

```
-r
AX=ØØØØ  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=ØØØØ  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø915  CS=Ø919  IP=ØØØØ   NV UP DI PL NZ NA PO NC
Ø919:ØØØØ 1E            PUSH    DS
```

The DS register contains 0905, so we know that in terms of absolute addresses the data segment starts at 09050 (at least for the moment; it will be changed soon, and then it will correspond with the figure). The stack segment starts at 09150, 100h bytes higher, and the code segment starts at 09190, 40h bytes above the start of the stack segment.

However, when PSTRING is executed it is going to change the value in the DS register to the segment address of the data segment, so we had better trace through these instructions in the program before we examine the contents of the DS register further.

Remember that it's important to check the value of the IP register before you begin to trace a program. If you have just loaded the progam, as we have here, IP will be automatically set to 0000 and there won't be any problem. However, if you have executed or traced any part of the program, you had better do an RIP and set IP back to zero.

Start tracing:

```
-t
AX=ØØØØ  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø915  CS=Ø919  IP=ØØØ1   NV UP DI PL NZ NA PO NC
Ø919:ØØØ1 2BCØ          SUB     AX,AX
```

Notice how the first PUSH instruction (shown in the last printout above) changes the Stack Pointer (SP) register to FFFE, at the high end of the stack segment.

```
-t
AX=ØØØØ  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=FFFE  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø915  CS=Ø919  IP=ØØØ3   NV UP DI PL ZR NA PE NC
Ø919:ØØØ3 5Ø            PUSH    AX
```

Now the program is going to take the segment address of the data segment and put it in the DS register. How does the program know what this address is? We certainly didn't know when we wrote the program. The answer is that this address is determined either by DOS or by

DEBUG, whichever is loading our program, *at the time the program is loaded.* Remember how the LST file showed these instructions?

```
                                        ;set DS register to current data
segment
ØØØ4  B8  ---- R                       mov  ax,datarea ;datarea segment addr
ØØØ7  8E D8                            mov  ds,ax      ;  into DS register
```

There is an " -- -- R" in the place where the segment address of the data segment (which is called "datarea" in this program) is supposed to be. When the program is loaded, DOS (or DEBUG) figures out what number should go here, and fills it into its proper place in the program code in memory. Let's watch what happens next:

```
-t
AX=ØØØØ  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø915  CS=Ø919  IP=ØØØ4   NV UP DI PL ZR NA PE NC
Ø919:ØØØ4 B815Ø9        MOV     AX,Ø915

-t
AX=Ø915  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=Ø915  CS=Ø919  IP=ØØØ7   NV UP DI PL ZR NA PE NC
Ø919:ØØØ7 8ED8          MOV     DS,AX


-t
AX=Ø915  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=FFFC  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø915  ES=Ø9Ø5  SS=Ø915  CS=Ø919  IP=ØØØ9   NV UP DI PL ZR NA PE NC
Ø919:ØØØ9 B931ØØ        MOV     CX,ØØ31
```

As you can see, the DS register now contains 0915, which is the address of the stack segment — just what we want.

> The DS and ES registers don't have the correct values until your program puts them there.

Now we can see that our data will start at absolute address 09150, and extend from there up to the beginning of the code part of our program at 09190, a difference of 40h bytes. Why 40h bytes? Because that's how much space our data needs: it's 31h bytes long, and since segments must start on "paragraph boundaries" — addresses which are multiples of 10h — the code segment starts 40h bytes beyond the start of

the data. The program instructions extend up to 09190h + 18h (the number of bytes in the program), which is 091A8.

The stack, on the other hand, is now at 09150h + FFFC, which is an absolute address of 1914C. This is almost 64K bytes above the program. The point is that there's plenty of room between the top of our program and the bottom of the stack, and that's why we can get away without specifying a stack segment. If we were writing larger programs, or using a lot of data, or were going to make extensive use of the stack, then we couldn't get away with this simple approach. Later we'll show you how to set up the stack segment when you need to.

How the Program Looks in Memory

Remember how the assembler could not generate the address of the data segment? This address will not be known until the program is loaded. If you want proof that DOS (or in this case DEBUG) can modify the code in our program and fill in this address, you can use the "U" command to list the program as it actually exists in memory:

```
-uØ,18
Ø919:ØØØØ 1E            PUSH    DS
Ø919:ØØØ1 2BCØ          SUB     AX,AX
Ø919:ØØØ3 5Ø            PUSH    AX
Ø919:ØØØ4 B8159Ø        MOV     AX,Ø915     ←— Address filled in
Ø919:ØØØ7 8ED8          MOV     DS,AX             by DEBUG
Ø919:ØØØ9 B931ØØ        MOV     CX,ØØ31
Ø919:ØØØC BBØØØØ        MOV     BX,ØØØØ
Ø919:ØØØF 8A17          MOV     DL,[BX]
Ø919:ØØ11 B4Ø2          MOV     AH,Ø2
Ø919:ØØ13 CD21          INT     21
Ø919:ØØ15 43            INC     BX
Ø919:ØØ16 E2F7          LOOP    ØØØF
Ø919:ØØ18 CB            RETF
```

Notice how, in the instruction at location 0004, the number 0915 (written reversed in the machine code as 1509) has actually been placed in its appropriate place in the instruction. Also notice that the program starts with an offset address of 0000 (the value in the IP register when we load the program) and a segment address of 0919 (the value we saw in the CS register). The "U" command in DEBUG thinks automatically in terms of the CS register.

Perhaps you're wondering why the lowest available segment address seems to be 0915h, at least on our particular PC system. The answer is that the DOS 2.00 operating system (which we're using for this example)

takes up somewhat less than 9150h bytes in low memory. Our program must therefore start just above that.

How the Data Look in Memory

We can also look at our data, using the "D" command, but be careful! It will not automatically think in terms of the DS register *until the program instructions that put the address of the data segment into the DS register have been executed.*

```
-dØ,3f
Ø915:ØØØØ  53 68 65 2Ø 69 73 2Ø 6D-6F 73 74 2Ø 66 61 69 72   She is most fair
Ø915:ØØ1Ø  2C 2Ø 74 68 6F 75 67 68-2Ø 73 68 65 2Ø 62 65 2Ø   , though she be
Ø915:ØØ2Ø  6D 61 72 62 6C 65 2D 68-65 61 72 74 65 64 2E ØD   marble-hearted..
Ø915:ØØ3Ø  ØA ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
```

Here the segment address is 0915, which is the address in the DS register. Since we traced through the opening instructions of the program, DS contains the correct value. If we had not executed these instructions we would have been looking at the wrong segment when we examined memory with "D," and who knows what we would have seen.

The OFFSET Operator

Did you notice the word "offset" in a line of code from the PSTRING program above?

```
mov  bx,offset string   :addr of string
```

What does this do? We're going to be using the BX register as a pointer to the various characters in the string. In other words, we're going to put the address of the first character into BX with the instruction shown above; then later we'll increment BX so it contains the addresses of other characters in the string, one after the other.

What would happen, however, if we wrote:

```
mov  bx,string      ;addr of string
```

leaving out the word offset? Instead of putting the *address* of the first character of the string into BX, we'd be putting *the character itself* into BX. That's no good — in fact, the assembler wouldn't even let us do it; it would see we were trying to put an 8-bit character into a 16-bit register and an assembly error would result.

What the word "offset" does is to tell the assembler that what we want

to put in BX is not the contents of "string", but the *address* of "string". The word "offset" in this case means the offset part of the address. (There is another, less commonly used operator called "seg", which performs a similar function for the segment part of the address.)

Thus, whenever we want the address of a variable in a program instruction, rather than the value of the variable, we must precede the variable name with the word "offset". You'll see this word used often in the programs that follow.

A Batch File for EXE Programs

When you learned about COM files we showed you how to construct a batch file to speed up the process of going from an ASM file to an executable program. Now we can do the same for EXE files. Here's a batch file which will transform your ASM file into an EXE file, leaving nothing else in your directory.

```
asm %1 %1 nul nul
link %1 @autolink
erase %1.bak
erase %1.obj
```

As before, the "autolink" file consists of nothing but three carriage return ↵ characters. Its name is simply "autolink," with no file extension.

You can call your batch file ASM2EXE.BAT for ASM to EXE. When executed it will erase the BAK file generated by your word processor and the OBJ file generated by the assembler, as well as create the EXE file.

The PIANO Program as an EXE File

Remember the trouble we had in the last chapter with the PIANO program? We had to assemble it twice: the first time just to figure out the address of our data (the table of frequencies), so that when we assembled it again we could use a hex address rather than a symbolic one. When we tried to use symbolic addresses in COM files we got into trouble.

Let's see how this program would look as an EXE file.

```
;PIANO--Uses Timer2 to run speaker
;  number keys play notes of the scale
;EXE version
```

```
= ØØ61                          portB   equ     61h     ;I/O Port B
= ØØØ7                          keybd2  equ     7h      ;keybd input, no echo
= ØØ21                          doscall equ     21h     ;DOS interrupt number
= ØØØ3                          cont_c  equ     Ø3h     ;control-C ASCII code

                                ;*********************************************

ØØØØ                            datarea segment         ;define data segment

                                ;frequencies of notes
ØØØØ  Ø1Ø6                      table   dw      262d    ;C
ØØØ2  Ø126                              dw      294d    ;D
ØØØ4  Ø14A                              dw      33Ød    ;E
ØØØ6  Ø15B                              dw      347d    ;F
ØØØ8  Ø188                              dw      392d    ;G
ØØØA  Ø1B8                              dw      44Ød    ;A
ØØØC  Ø1EE                              dw      494d    ;B
ØØØE  Ø2ØC                              dw      524d    ;C

ØØ1Ø                            datarea ends
                                ;*********************************************

ØØØØ                            prognam segment         ;define code segment

                                ;---------------------------------------------
ØØØØ                            main    proc    far     ;main part of program

                                        assume  cs:prognam,ds:datarea

ØØØØ                            start:          ;starting execution address

                                ;set up stack for return
ØØØØ  1E                               push ds          ;save old data segment
ØØØ1  2B CØ                            sub  ax,ax       ;put zero in AX
ØØØ3  5Ø                               push ax          ;save it on stack


                                ;set DS register to current data segment
ØØØ4  B8  ---- R                       mov  ax,datarea ;datarea segment addr
ØØØ7  8E D8                            mov  ds,ax      ;  into DS register

                                ;read keyboard to get digit from Ø to 7
ØØØ9                            read_key:
ØØØ9  B4 Ø7                            mov   ah,keybd2 ;keybd funct, no echo
ØØØB  CD 21                            int   doscall   ;call DOS
ØØØD  3C Ø3                            cmp   al,cont_c ;is it control-C ?
ØØØF  74 34                            jz    exit      ;yes, so exit
Ø011  2C 31                            sub   al,31h    ;change ASCII to digit
Ø013  24 Ø7                            and al,ØØØØØ111b ;mask off hi 5 bits
```

```
0015  D0 E0                         shl   al,1       ;* by 2 (2 bytes/word)
0017  98                            cbw             ;byte --> word in AX
0018  8B D8                         mov   bx,ax     ;put in bx (for table)
001A  B8 0000                       mov   ax,0      ;numerator (low byte)
001D  BA 0012                       mov   dx,12h    ;  (high byte)
0020  F7 B7 0000 R                  div   [table + bx] ;divisor from table
0024  8B D8                         mov   bx,ax     ;save quotient in BX

                              ;set 1/pitch into timer, then turn on tone
0026  B0 B6                         mov   al,10110110b  ;put magic number
0028  E6 43                         out   43h,al    ;     into timer2
002A  8B C3                         mov   ax,bx     ;1/pitch into AX
002C  E6 42                         out   42h,al    ;LSB into timer2
002E  8A C4                         mov   al,ah     ;MSB to AL, then
0030  E6 42                         out   42h,al    ;  to timer2
0032  E4 61                         in    al,portB  ;read port B into AL
0034  0C 03                         or    al,3      ;turn on bits 0 and 1
0036  E6 61                         out   portB,al  ;to turn on speaker


                              ;sound note for a while, then turn it off
0038  B9 FFFF                       mov   cx,0ffffh ;set up for delay
003B  E2 FE                   wait: loop  wait      ;delay
003D  E4 61                         in    al,portB  ;read port B into AL
003F  24 FC                         and   al,11111100b ;mask lower 2 bits
0041  E6 61                         out   portB,al  ;to turn off speaker
0043  EB C4                         jmp   read_key  ;go get another digit


                              ;control-C typed, so exit
0045                          exit:
0045  CB                            ret             ;return to DOS


0046                          main    endp    ;end of main part of program
                              ;-----------------------------------------
0046                          prognam ends    ;end of code segment
                              ;*****************************************

                                      end     start   ;end assembly
```

There are several things to notice about this program. First, we've given the first location of our data the name "table." Again, this name, being in the data segment (not an address in the code segment), is *not* followed by a semicolon.

Second, we've used this symbolic address in the DIV instruction at location 0020:

```
0020  F7 B7 0000 R            div   [table + bx] ;divisor from table
```

This instruction takes the content of the BX register and adds it to the offset address of "table" (which happens to be 0000, since "table" is the first item in the data segment), and uses the result as its EA (effective address). It then takes the word from this address and divides it into the double-word DX+AX. The important point here is that we are able to use a symbolic name for "table," rather than a number as we did in the COM file version of this program.

The "R" following the hex op-code in this instruction is a warning that the instruction references a segment outside the code segment. Programs that contain instructions that have this "R" (for "relocatable") in the LST file will not work as COM files — this is the giveaway that an EXE file is called for.

The EXEFORM Program — A Nonprogram

As you've noticed in the PSTRING and PIANO programs, EXE files have a fairly large amount of "overhead" in the form of program statements which must always be in the program, no matter what the program is supposed to do. If your word processor has the capability of reading one text file into another one (and it should), you can cut down a lot of the work of writing a program by setting up all this "overhead code" in a separate file. Call this file EXEFORM.ASM (for "EXE file FORMat"). Then, whenever you want to write a program that will become an EXE file, you can start off with this file, and fill in the program instructions and data in the appropriate places.

Let's look at the LST file version of EXEFORM:

```
                        ;PROGRAM TITLE GOES HERE--
                        ;  Followed by descriptive phrases

                        ;EQU STATEMENTS GO HERE

                        ;*********************************************

0000                    datarea segment         ;define data segment

                        ;DATA GOES HERE

0000                    datarea ends
                        ;*********************************************

0000                    prognam segment         ;define code segment

                        ;---------------------------------------
```

```
0000                    main    proc    far     ;main part of program

                                assume  cs:prognam,ds:datarea

0000                    start:          ;starting execution address

                        ;set up stack for return
0000  1E                        push ds         ;save old data segment
0001  2B C0                     sub  ax,ax      ;put zero in AX
0003  50                        push ax         ;save it on stack

                        ;set DS register to current data segment
0004  B8  ---- R                mov  ax,datarea ;datarea segment addr
0007  8E D8                     mov  ds,ax      ;  into DS register

                        ;MAIN PART OF PROGRAM GOES HERE

0009  CB                        ret             ;return to DOS

000A                    main    endp    ;end of main part of program
                        ;-----------------------------------------
000A                    subr1   proc    near    ;define subprocedure

                        ;SUBROUTINE GOES HERE

000A                    subr1   endp            ;end subprocedure
                        ;-----------------------------------------
000A                    prognam ends    ;end of code segment
                        ;*****************************************

                                end     start   ;end assembly
```

This is the bare skeleton of an EXE file. You can actually assemble, link, and execute this program, and it will return properly to DOS. Of course, that's *all* it will do: there are no instructions to make it do anything else. Its purpose is to serve as a sort of "template" for other programs. Type in the right-hand side of the listing, and save it on your disk as EXEFORM.ASM. Then whenever you're ready to write an EXE program, you can start with EXEFORM, fill in the 8088 instructions to make the program do what you want, and you're off and running.

You'll notice that EXEFORM includes space for a subroutine — actually a NEAR PROCedure. This procedure can be called from the main program with a CALL instruction, as we've seen before. You can add as many of these NEAR PROCedures in a program as you want. You should be familiar with these NEAR PROCedures from the DECIHEX program in chapter 6.

Using the Stack Segment

It is probably somewhat irresponsible for us to give the impression that it is good programming practice to ignore the stack segment as we have been doing up to now. In fact, it's much safer to set up a separate stack segment in your program. Then you know where it's supposed to be, and how big it is, and you can examine it more easily when you're debugging.

> You really should always use a stack segment.

How do you set up a stack segment in your program? We've organized another form of the EXEFORM program which includes this feature, as shown in the listing below. (In the interests of conserving space we took out the subroutine at the end of this program. You can put it back in if you need it.) Here's the listing:

```
                        ;PROGRAM TITLE GOES HERE--
                        ;  Followed by descriptive phrases

                        ;EQU STATEMENTS CAN GO HERE

                        ;*********************************************

0000                    st_seg  segment  stack   ;define stack segment
0000    14 [                    db       20 dup  ('stack   ')
           73 74 61 63
           6B 20 20 20
                        ]
00A0                    st_seg  ends

                        ;*********************************************

0000                    datarea segment         ;define data segment

                        ;DATA CAN GO HERE

0000                    datarea ends
                        ;*********************************************

0000                    prognam segment         ;define code segment

                        ;---------------------------------------
0000                    main    proc    far     ;main part of program
```

```
                                assume  cs:prognam,ds:datarea

0000                    start:          ;starting execution address

                        ;set up stack for return
0000  1E                        push ds         ;save old data segment
0001  2B C0                     sub  ax,ax      ;put zero in AX
0003  50                        push ax         ;save it on stack

                        ;set DS register to current data segment
0004  B8  ---- R                mov  ax,datarea ;datarea segment addr
0007  8E D8                     mov  ds,ax      ;  into DS register

                        ;MAIN PART OF PROGRAM CAN GO HERE

0009  CB                        ret             ;return to DOS

000A                    main    endp    ;end of main part of program
                        ;----------------------------------------
000A                    prognam ends    ;end of code segment
                        ;****************************************

                                end     start   ;end assembly
```

As you can see, an addition has been made to the *segment* pseudo-op to define the stack segment, and a new pseudo-op, DUP, has been used in this segment. We'll cover these two items separately below.

The Combine-type Entry in the Segment Pseudo-op

We defined the stack segment in the usual way with a SEGMENT pseudo-op, but we also added something: the word STACK in the operand field. What does this do? Its purpose is to tell the linker that this segment is in fact a stack segment. One result of this is that the error message "Warning, no stack segment," which the linker has been bothering us with chapter after chapter, will now disappear.

The more significant effect is that, if we were using the linker to combine several programs, the linker would know that this segment (ST_SEG) should be combined with other stack segments into a single stack segment. Also, when our program is loaded, DOS (or DEBUG) will make sure that the Stack Segment register (SS) is loaded with the appropriate address.

The word STACK used in this way in the SEGMENT pseudo-op is an example of something called a "combine-type." As you can see by looking up SEGMENT in the section on pseudo-ops in the *IBM Personal Computer MACRO Assembler* manual, there are several other combine-

types: PUBLIC, COMMON, AT expression, and MEMORY. We'll be covering some of these later on in the book, but for the moment you don't need to know about them.

The Align-type Entry in the Segment Pseudo-op

While we're talking about the SEGMENT pseudo-op, we might also mention the "align-type" entry. This is an entry (a word) which precedes the combine-type in the segment pseudo-op. Its purpose is to tell the linker on what sort of address boundary the segment should start. The segment can start on a byte, which means any address; it can start on a word, which means an even-numbered address; it can start on a "paragraph," which is an address divisible by 10h (16d); or it can start on a page, which is an address divisible by 100h (256d). The words used to select these starting address boundaries are BYTE, WORD, PARA and PAGE, respectively.

Usually we want the segment to start on a paragraph boundary; so we want the PARA align-type. However, PARA is the default align-type, so if we don't use any of these words, PARA is what we'll get. That's why we don't usually need to use the align-type entry in this pseudo-op.

There can also be a "class" entry in the SEGMENT pseudo-op. This is a name, surrounded by single quotes, that is used by the linker to combine together segments which are in different programs but which have the same class name.

An example of a segment pseudo-op specifying a data segment starting on a word boundary and having the combine-type entry "public" and the class 'zip_codes' would be

```
data_seg    segment   word   public  'zip_codes'
```

For the time being you don't have to worry about any of this except the STACK combine-type.

The DUP Expression

DUP is not a pseudo-op itself, but is used as an optional expression in pseudo-ops like DB (Define Byte) and DW (Define Word). Its purpose is to make duplicate copies of whatever follows it, enclosed in parentheses, in the operand field of the pseudo-op. The number preceding the DUP tells how many copies are to be made. Thus in the example from the EXEFORM program above, the DB pseudo-op specifies that 20d copies of the string 'stack' (that's "stack" followed by three spaces) are to be placed in memory. You can see how the LST file displays the resulting hex characters.

```
0000     14 [                        db       20 dup ('stack   ')
               73 74 61 63
               6B 20 20 20
                           ]
```

As IBM suggests in the program listing in the *IBM Personal Computer MACRO Assembler* manual (appendix D), we have caused the stack to be filled in with the string "stack ", which is 8 characters long. Why do this? Because it makes it very clear where the stack is when you go to debug your program. Once you find the stack, you can see at a glance how much of it has been used: the parts *not* used are filled with the word "stack", which is readily visible with the "D" command.

Let's see how this might look. Assemble and link the EXEFORM2 program into an EXE file. Then load it, using DEBUG:

```
A>debug exeform2.exe

-r
AX=0000  BX=0000  CX=0100  DX=0000  SP=00A0  BP=0000  SI=0000  DI=0000
DS=0905  ES=0905  SS=0916  CS=0915  IP=0000   NV UP DI PL NZ NA PO NC
0915:0000 1E            PUSH    DS
```

Notice that the SP register no longer says 0000 as it does when we *don't* define a stack segment. It has been set to 00A0, which is just large enough to hold the 20d copies of the string "stack ", which we specified were to go in the stack segment. Now we'll examine the stack segment:

```
-d916:0
0916:0000  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0010  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0020  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0030  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0040  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0050  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0060  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0070  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
-d
0916:0080  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 20 20   stack   stack
0916:0090  73 74 61 63 6B 20 20 20-73 74 61 63 6B 20 00 00   stack   stack ..
0916:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
0916:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

You can see why we wanted to DUPlicate an 8-character string: it fits so neatly on the 16-character line. The stack will grow downward from 0916:00A0. In this case the first two locations, at 009E and 009F, have already been filled in with zeros.

Segmentation and the String-Handling Instructions

The 8088 contains some rather special instructions for handling strings. In this case "strings" means any sequences of bytes or words stored in memory, whether they are ASCII characters or not. These string-handling instructions are closely related to the idea of memory segmentation, and in particular to the use of the Extra Segment. We're going explain how these instructions work, and then show you a program that uses these string instructions to compare two strings, and at the same time makes use of the Extra Segment, the Data Segment, the Stack Segment, and the Code Segment. It will be your first view of a program making use of all four segments.

Of course the string-handling instructions can be used for purposes other than text manipulation. In particular, they're useful for *graphics*, where their great speed can be used to do such things as moving an entire graphics image into the screen memory — a process important in animation. We'll be talking more about graphics in chapter 10.

String-Handling Instructions

The string-handling instructions make use of a unique idea in the 8088 instruction set: the REP instruction. REP stands for "REPeat," and what it does is to *cause the instruction which follows it to be repeated until CX becomes zero.* REP is used with a special group of three instructions which can move a string from one place to another in memory, scan a string for a particular byte or word, and compare one string with another. These instructions are called MOVS (MOVe String), SCAS (SCAn String), and CMPS (CoMPare String).

The REP Instruction

REP Instruction

Causes following string instruction to repeat.

The string instruction following REP will be repeated until CX becomes zero.

Flags affected: depends on string operation.

The SI and DI Registers

The string instructions make use of two registers which we haven't talked about yet: the SI and DI registers. You can see them in the upper

left-hand corner of the printout when you use the "R" command in DEBUG. SI stands for "Source Index," and DI stands for "Destination Index." These registers are used as *pointers* or *indexes*, to point to the strings to be operated on in memory. For instance, suppose we wanted to MOVe a string of bytes from one part of memory to another. We'd set the SI register to point to the start of the string we wanted to move, and we'd set the DI register to point to the first location of the area we wanted to move the string to. This operation is shown in Figure 8-4.

Data Segment and Extra Segment

Now, here's where the segments fit into the string handling instructions: the DI register is assumed to contain an address *which is in the Data Segment*, and the SI register is assumed to contain an address *which is in the Extra Segment.* Thus, before you can use the string-handling instructions, you have to do the following things:

1. Set up a buffer (an area of memory) for one string in the data segment.
2. Use an ASSUME statement to tell the assembler that the data segment will be in the DS register.
3. Put the segment address of the data segment into the DS register.
4. Put the offset address of the source string into the SI register.
5. Set up a buffer for the second string (or the place the first string

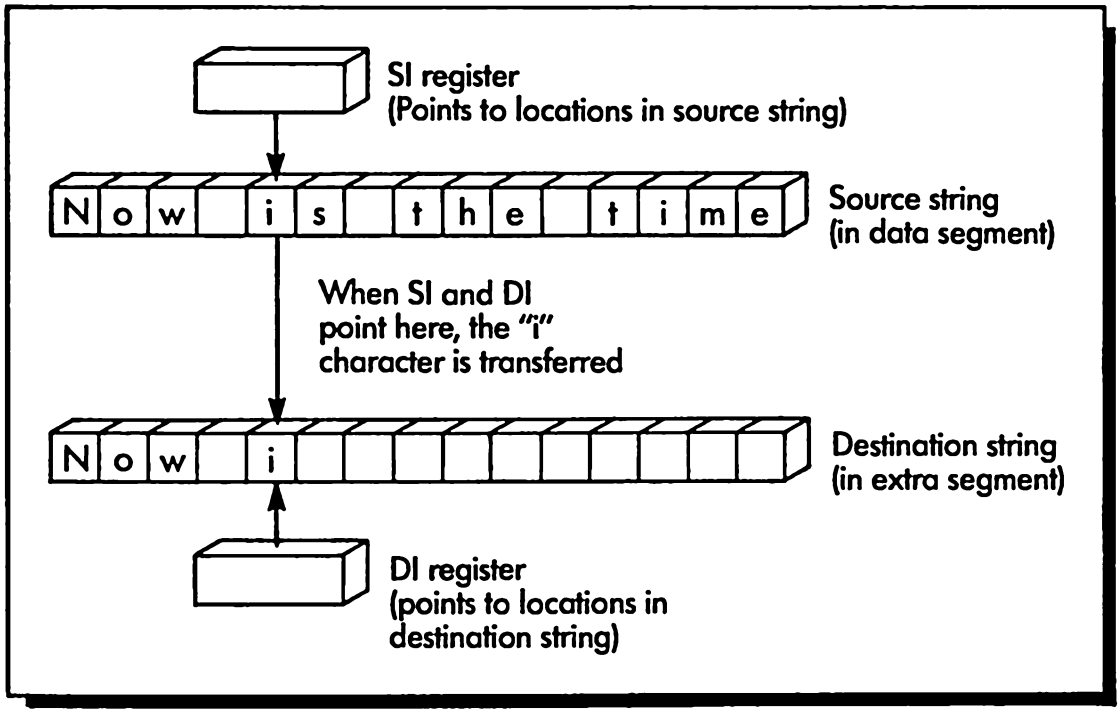


Figure 8-4. Use of SI and DI registers for string instructions

will be moved to) in the extra segment.

6. Use an ASSUME statement to tell the assembler that the extra segment will be in the ES register.

7. Put the segment address of the extra segment into the ES register.

8. Put the offset address of the destination string into the DI register.

Also, we need to put the count of how many characters are to be moved, scanned, or compared into the CX register.

> The source buffer for string operations goes in the Data Segment, the destination buffer in the Extra Segment.

The MOVS Instruction

Strings are moved from one place to another in memory using the REP instruction followed by the MOVS instruction.

> ## MOVS Instruction
>
> MOVes a String from one location to another.
>
> Used following a REP instruction. The string in the data segment, pointed to by SI, is transferred to the buffer in the extra segment, pointed to by DI.
>
> The number of bytes to be moved is specified by the CX register.
>
> Each time MOVS is executed:
>
> 1. SI is incremented (or decremented, depending on direction flag), either one or two bytes (depending on whether bytes or words are being transferred).
> 2. DI is incremented (or decremented, depending on direction flag), either one or two bytes (depending on whether bytes or words are being transferred).
> 3. CX is decremented. If it becomes zero, transfer is terminated.
>
> Flags affected: none

Shown below is some code that moves a string of 40 letter "a"s from a source buffer to a destination buffer. In the interest of brevity this is not a complete program. Only those instructions relating to the string-handling instructions are shown.

We'll be using the string-handling instruction MOVS in this code fragment. There are two ways to put the RET instruction together with string-handling instructions like MOVS. The simplest, and the one we'll show you here, is to put them on the same line.

```
;**************************************
data_seg  segment
source_buffer db 40 dup ('a')
data_seg  ends
;**************************************
extra_seg segment
dest_buffer   db 40 dup (?)
extra_seg ends
;**************************************

        assume ds:data_seg          ;tell assembler where
        assume es:extra_seg         ;   segments are

        mov  ax,data_seg            ;put segment address of
        mov  ds,ax                  ;  source buffer in DS

        mov  ax,extra_seg           ;put segment address of
        mov  es,ax                  ;  dest buffer in ES

        mov  si, offset source_buffer ;put offset address of
                                    ;  source buffer in SI

        mov  di, offset dest_buffer ;put offset address of
                                    ;  dest buffer in DI

        cld                         ;set direction flag
                                    ;  to forward

        mov  cx,count               ;put count in CX

        rep  movs                   ;move entire string!
```

As you can see, there's a lot of setting up, but once it's done, the actual transfer of the data only takes two instructions, written on one line! They are each one-byte instructions, too, so the transfer goes very fast.

How does it work? The REP instruction causes the MOVS instruction to be executed over and over until CX becomes zero. Each time the

MOVS instruction is executed it transfers a byte from the source buffer — at the location pointed to by SI; to the destination buffer — to the location pointed to by DI. It also increments the DI and SI registers to point to the *next* byte to be transferred and the *next* location it's to be transferred to. MOVS also decrements the CX register. If CX becomes zero, the process stops, and the program goes on to the next instruction following the MOVS.

The CLD Instruction

We've used another instruction you haven't seen before, called CLD. Remember back when we talked about flags, there was something called the "direction flag"? The direction flag is used to specify whether SI and DI will be automatically *incremented*, which has been our assumption in this discussion, or whether they will be *decremented*. CLD stands for CLear Direction flag, and clearing this flag specifies the "forward" direction, which means the pointers will be automatically incremented. If we had used the instruction STD — which means SeT Direction flag — the flag would have been set to "backwards" and these registers would have been automatically *decremented*. In that case we would have had to set SI and DI to the *ends* of their respective strings, rather than the beginnings. As you can see, the forward direction is more natural and is therefore more commonly used, so in general a CLD instruction will precede your string-handling instructions.

CLD Instruction

Sets Direction Flag to forward direction (CLD stands for "CLear Direction flag").

Used to specify that SI and DI will be automatically incremented during string-handling instructions.

STD — (for "SeT Direction flag") is used to specify that SI and DI will be decremented.

Flags affected: DF

The Compare Strings Program

In the next program we use the string-handling instruction CMPS to compare two strings. CMPS is similar to MOVS in the way it operates and

in how you need to set things up before you use it. However, instead of MOVing a string from one buffer to another, it CoMPares a string in one buffer with a string in another buffer.

The CMPS Instruction

CMPS Instruction

CoMPares a String in one location to a string in another.

Used following a REPE instruction. The string in the data segment, pointed to by SI, is compared to the string in the extra segment, pointed to by DI.

The number of bytes to be compared is specified by the CX register.

Each time CMPS is executed:

1. SI is incremented (or decremented, depending on direction flag), either one or two bytes (depending on whether bytes or words are being transferred).
2. DI is incremented (or decremented, depending on direction flag), either one or two bytes (depending on whether bytes or words are being transferred).
3. CX is decremented. If it becomes zero, the process is terminated.

Flags affected: AF, CF, OF, PF, SF, 2F

You may have asked yourself this question: How do we know, when the comparison operation is over, whether the two strings we just compared were in fact the same? We use a variation of the REPeat instruction, called REPE, which means "REPeat while Equal." This provides a way of terminating the comparison procedure *before* the CX register becomes zero. In a comparison, we generally want to stop the comparison process as soon as we realize that the comparison is failing, that is, when we discover that a character in the first string and the corresponding character in the second string are not the same. REPE will permit the CMPS instruction to be executed only as long as the results of the comparison are equal. Once a nonmatch is found, the program goes on to the instruction following the CMPS.

If the comparison does fail, so that REPE causes an early termination of the process, *the CX register will not yet be zero.* We can determine this fact using a JZ or JNZ instruction, since if CX had not become zero, the zero flag will not be set. This is how we find out the results of the comparison operation.

Thus, our strategy is to perform the comparison by executing the REPE CMPS instruction combination, and then checking the zero flag. If it is zero, then all the bytes checked passed the comparison check, and the two strings are equal. If it is not zero, then the comparison terminated early, CX is not zero, and the two strings do not match. If we want, we can find out the location where the match fails by looking in the SI or DI registers, or by examining the count left in CX.

The REPE and REPZ Instructions

REPE Instruction

Causes the CMPS or SCAS instruction that follows to be repeated (REPE stands for "REPeat while Equal").

The REPE and CMPS (or SCAS) instructions will be repeated until either CX becomes zero, or the results of a comparison cause the zero flag to be cleared (set to NZ), as the result of two bytes (or words) *not* matching.

This instruction can also be written REPZ, for "REPeat while Zero."

Flags affected: depends on string operation

The SCAS Instruction

There is another instruction, SCAS, which we won't describe in detail but which we'll mention to complete the list of string-handling instructions. It stands for "SCAn String," and (unlike MOVS and CMPS) operates on only *one* string, in the DS segment, pointed to by the SI register. The contents of the AL register (or AX, if words are being scanned for) are compared with the succesive bytes in the string. Like CMPS, SCAS can be used with REP and REPE. It's probably most often used with another instruction, REPNE, which stands for "REPeat while Not Equal." SCAS scans through a string looking for a particular byte (or

word). If it finds it, we want to terminate the scan. That's why we use REPNE: as soon as a *matching* character is found, the search is ended, even though CX is not zero.

The REPNE and REPNZ Instructions

REPNE Instruction

Causes the CMPS or SCAS instruction that follows to be repeated (REPNE stands for "REPeat while Not Equal").

The REPE and CMPS (or SCAS) instructions will be repeated until either CX becomes zero, or the results of a comparison cause the zero flag to be set (cleared to ZE), as the result of two bytes (or words) matching.

This instruction can also be written REPNZ, for "REPeat while Not Zero."

Flags affected: depends on string operation.

The SEARCH Program Listing

The following program uses the REPE and CMPS instructions to compare a "keyword," typed in by the user, to a "sentence," typed in following the keyword. If the keyword is contained in the sentence, the program prints "Match!!!" If the sentence does not contain the word, the program prints "No match." We've made this a complete program so you can see how the various parts all work together. The listing is fairly long, but it's divided into sections which individually are short and not difficult to understand.

```
                        ;SEARCH--Searches one string for another
                        ;  User types in keyword, then sentence
                        ;  Program decides if sentence contains word
                        ;  Prints out conclusion

= ØØØ9                  print_m equ 9h  ;print string function
= ØØØA                  buff_in equ Øah ;buffered kbd input function
= ØØ21                  doscall equ 21h ;DOS interrupt number

                        ;*****************************************
```

```
0000                            st_seg  segment  stack   ;define stack segment
0000    14 [                            db       20 dup  ('stack   ')
           73 74 61 63
           6B 20 20 20
                        ]

00A0                            st_seg  ends

                                ;*********************************************

0000                            datarea segment          ;define data segment

                                ;buffer to hold sentence
0000  7F                        sen_max  db 127d         ;max chars in sentence
0001  ??                        sen_real db ?            ;actual chars in sent.
0002    7F [                    sentence db 127d dup (?) ;space for 127 chars
            ??
                ]


                                ;buffer to hold keyword
0081  14                        key_max  db 20d          ;max chars in keyword
0082  ??                        key_real db ?            ;actual chars in keywd
0083    14 [                    keyword  db 20d dup (?) ;space for 20 chars
            ??
                ]


0097  0D 0A 45 6E 74 65        mess1   db      0dh,0ah,'Enter Keyword: $'
      72 20 4B 65 79 77
      6F 72 64 3A 20 24
00A9  0D 0A 45 6E 74 65        mess2   db      0dh,0ah,'Enter Sentence: $'
      72 20 53 65 6E 74
      65 6E 63 65 3A 20
      24
00BC  0D 0A 4E 6F 20 6D        mess3   db      0dh,0ah,'No match.$'
      61 74 63 68 2E 24
00C8  0D 0A 4D 61 74 63        mess4   db      0dh,0ah,'Match!!!$'
      68 21 21 21 24

00D3                            datarea ends
                                ;*********************************************

0000                            prognam segment          ;define code segment

                                ;-----------------------------------------
0000                            main    proc    far      ;main part of program
```

```
                                assume  cs:prognam,ds:datarea
                                assume  es:datarea

0000                    start:          ;starting execution address

                        ;set up stack for return
0000  1E                        push ds         ;save old data segment
0001  2B C0                     sub  ax,ax      ;put zero in AX
0003  50                        push ax         ;save it on stack


                        ;set DS register to current data segment
0004  B8  ---- R                mov  ax,datarea ;datarea segment addr
0007  8E D8                     mov  ds,ax      ;  into DS register

                        ;set ES register to current extra segment
0009  B8  ---- R                mov  ax,datarea ;datarea segment addr
000C  8E C0                     mov  es,ax      ;  into ES register

                        ;GET KEYWORD AND PUT IN BUFFER
000E                    new_key:

                        ;print "enter keyword" message
000E  BA 0097 R                 mov  dx,offset mess1 ;addr in DX
0011  B4 09                     mov  ah,print_m ;print string function
0013  CD 21                     int  doscall    ;call DOS

                        ;get keyword and put in buffer
0015  BA 0081 R                 mov  dx,offset key_max ;addr of buffer

0018  B4 0A                     mov  ah,buff_in ;buffered keybd input
001A  CD 21                     int  doscall    ;call DOS

                        ;GET SENTENCE AND PUT IN BUFFER
001C                    new_sent:

                        ;print "enter sentence" message
001C  BA 00A9 R                 mov  dx,offset mess2 ;addr in DX
001F  B4 09                     mov  ah,print_m ;print string function
0021  CD 21                     int  doscall    ;call DOS

                        ;get sentence and put in buffer
0023  BA 0000 R                 mov  dx,offset sen_max ;addr of buffer

0026  B4 0A                     mov  ah,buff_in ;buffered keybd input
0028  CD 21                     int  doscall    ;call DOS
```

```
                        ;SEARCH FOR KEYWORD IN SENTENCE

                        ;SI register holds pointer to keyword
                        ;DI register holds pointer to sentence
                        ;BX register holds pointer to
                        ;       current starting place in sentence
                        ;DX register holds count of chars in sentnece
                        ;       less chars in keyword + 1
                        ;CX register holds count of chars in word

ØØ2A  FC                        cld  ;set direction flag to forward

                        ;calculate length of sentence less
                        ;  length of keyword, put in DX
ØØ2B  AØ ØØØ1 R                 mov  al,sen_real ;length of sentence
ØØ2E  2A Ø6 ØØ82 R              sub  al,key_real ;less length of word
ØØ32  7C 25                     jl   no_match    ;word longer than s.
ØØ34  98                        cbw              ;change byte to word
ØØ35  8B DØ                     mov  dx,ax       ;put in DX
ØØ37  42                        inc  dx          ; + 1

                        ;set BX to first character in sentence
ØØ38  BB ØØØ2 R                 mov  bx,offset sentence

ØØ3B                    compare:

                        ;set DI to BX--this is place in sentence
                        ;  where comparison will begin
ØØ3B  8B FB                     mov  di,bx

                        ;set SI to start of keyword
ØØ3D  BE ØØ83 R                 mov  si,offset keyword

                        ;set CX to number of characters in keyword
ØØ4Ø  AØ ØØ82 R                 mov  al,key_real ;get count
ØØ43  98                        cbw              ;change byte to word
ØØ44  8B C8                     mov  cx,ax       ;put in CX

                        ;compare keyword to this part of sentence
ØØ46  F3/ A6                    repe cmpsb       ;compare characters
                                                 ;repeat until CX = Ø
                                                 ;or nonmatch is found
ØØ48  74 Ø6                     jz   match       ;match found

                        ;no match found here.  Advance BX to next
                        ;  character in sentence, check if done
ØØ4A  43                        inc  bx          ;advance pointer
```

```
ØØ4B  4A                          dec  dx          ;done?
ØØ4C  74 ØB                       jz   no_match    ;yes, no match
ØØ4E  EB EB                       jmp  compare     ;no, try again



                                  ;print "match" message
ØØ5Ø                              match:
ØØ5Ø  BA ØØC8 R                          mov  dx,offset mess4 ;addr in DX
ØØ53  B4 Ø9                              mov  ah,print_m ;print string function
ØØ55  CD 21                              int  doscall     ;call DOS
ØØ57  EB C3                              jmp  new_sent    ;get another sentence

                                  ;print "no match"  message
ØØ59                              no_match:
ØØ59  BA ØØBC R                          mov  dx,offset mess3 ;addr in DX
ØØ5C  B4 Ø9                              mov  ah,print_m ;print string function
ØØ5E  CD 21                              int  doscall     ;call DOS
ØØ6Ø  EB BA                              jmp  new_sent    ;get another sentence


ØØ62  CB                                 ret              ;return to DOS


ØØ63                              main    endp     ;end of main part of program
                                  ;-----------------------------------------
ØØ63                              prognam ends     ;end of code segment
                                  ;*****************************************
                                  ;
                                          end      start   ;end assembly
```

The flow chart in Figure 8-5 shows the overall operation of the SEARCH program.

One important thing to notice in the SEARCH program is that *it uses the same segment for both the data segment and the extra segment.* There's no reason why the program can't do this, as long as it tells the assembler what it's doing, using the ASSUME statements; and puts the proper segment address into the DS and ES registers, as shown in locations 0004 through 000C.

The result is that the extra segment is the same segment as the data segment. This makes things a little easier when we're using string-handling instructions, since we don't have to remember in which segment a particular variable or buffer is: it's always in "datarea," whether it's in the data segment or the extra segment.

The printing of messages and the reading of strings from the

keyboard should be familiar to you. We've used the Print String and Buffered Keyboard Input DOS functions before. Notice again how the word "offset" is used to specify the address of a message or buffer, rather than the data at that address.

Once the keyword and the sentence are safely stored in their buffers, the program can proceed to the actual comparison. It uses the BX register to hold a pointer to the characters in the sentence buffer. The comparison starts on the first character. Let's assume that our keyword is three letters long. A comparison is then made between the keyword and

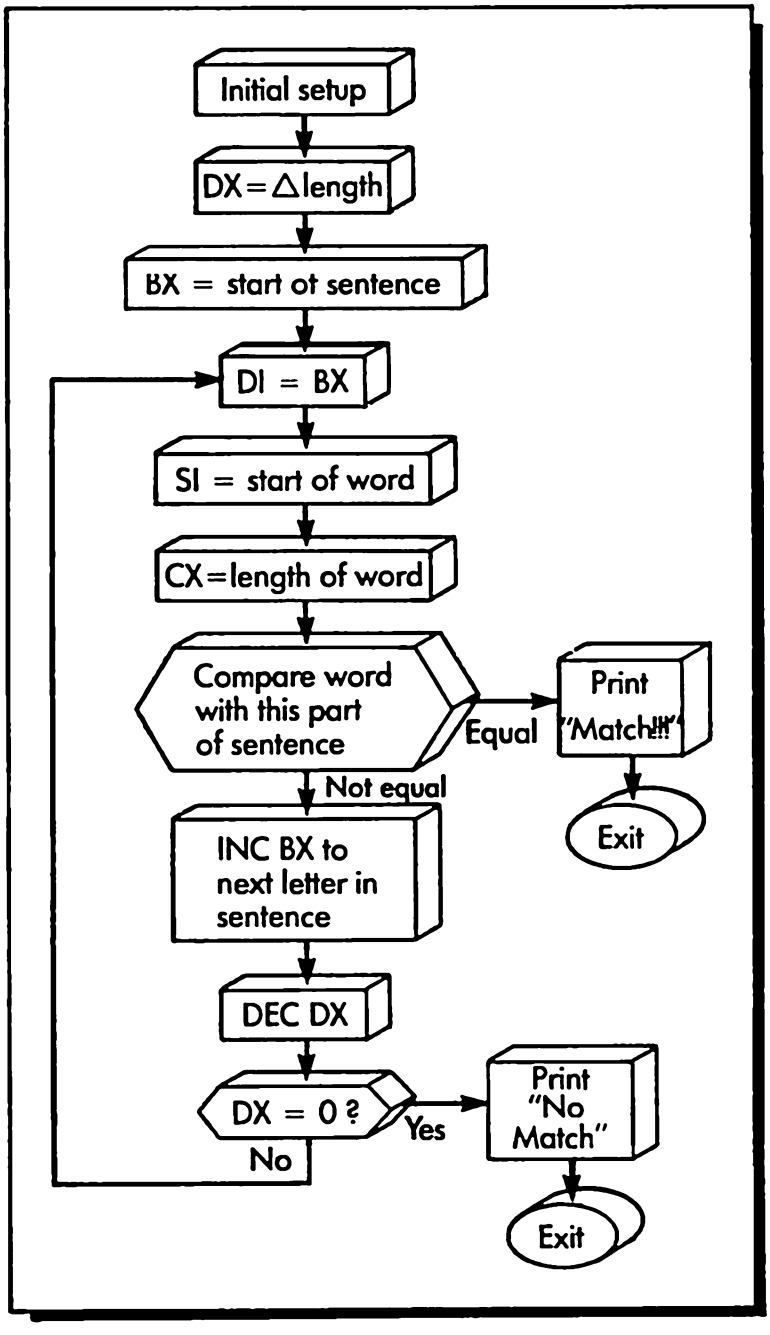


Figure 8-5. Flow chart of the string search program

the first three letters of the sentence. This comparison is made as follows:

1. The SI register points to the letters in the keyword.
2. The DI register points to the letters in the sentence.
3. The CX register holds the count of the number of letters in the keyword.

Figure 8-6 shows how the registers are used as pointers.

The CMPS instruction is repeated until either CX becomes zero, or until the comparison fails. If CX becomes zero before the comparison fails, we know we have a match, so we jump to the section of the program which prints out "Match!!!" If the comparison fails, we increment BX to point to the second letter in the sentence, and test letters 2, 3, and 4 against the keyword.

How does the program know when to stop? Suppose the sentence is 10 letters long, and the keyword is 3. There's no point in testing closer to the end of the sentence than the 8th letter. For instance, don't try to match 9,10,11 against 1,2,3, because there is no 11th letter in the sentence. The number of letters in the sentence we want to check is thus:

10 − 3 + 1 = 8.

In general the number of letters we want to check is equal to the length of the sentence, less the length of the keyword, plus 1. The program calculates this number in locations 002B to 0037. One of the nice things about the Buffered Keyboard Input DOS function is that it

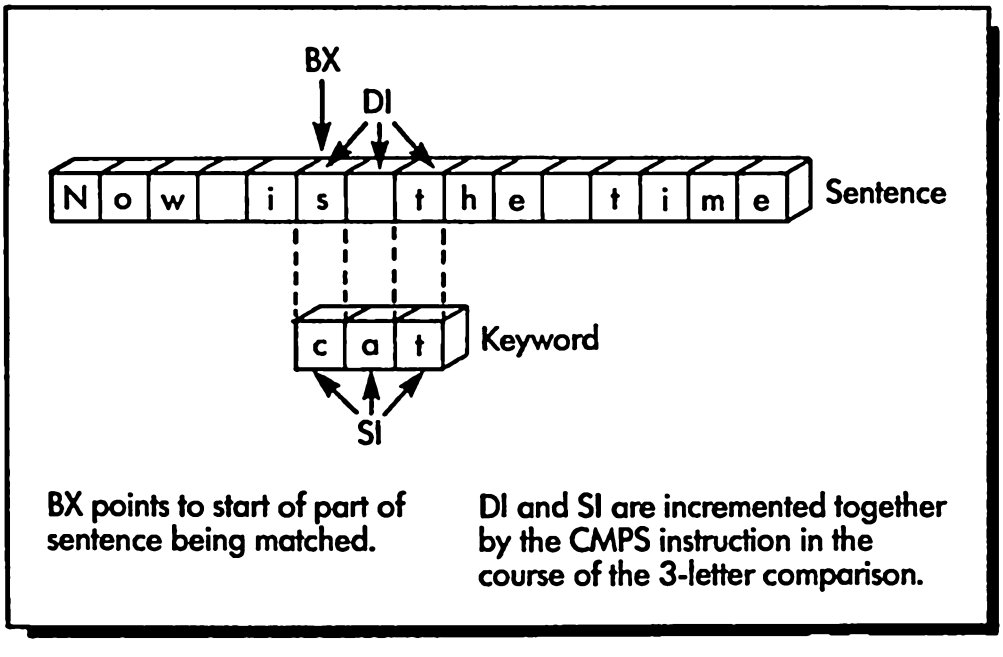


Figure 8-6. Use of pointers in the string search program

returns these lengths for us, in the "sen_real" and "key_real" locations immediately preceding the buffers, so we don't have to calculate them ourselves.

Type in this program, assemble it, and try it out. It's kind of fun watching it work, and knowing that only two short instructions are involved in the actual comparison process.

Summary

In this chapter you've learned about memory segmentation and about EXE files. You were also introduced to the 8088's string-handling instructions, which gave you a chance to see one of the common applications of the segment registers, in particular the extra segment. We'll use the segment registers again very soon, in a slightly different way.

9

Inside the ROM

Concepts

The ROM BIOS
The ROM BIOS listing
Keyboard scan codes
Keyboard shift status
The monochrome screen
Video character attributes

8088 Instructions

INT = Interrupt
IRET = Interrupt return

ROM BIOS Functions

Keyboard
    Return ASCII and Scan Code
    Return Shift Status
Video
    Clear Screen
    Scroll Active Page Up
    Set Cursor Position

The question "What is ROM?" sounds like it might be asked in a Zen monastery about an obscure point of religious doctrine, but in our case it has a very specific answer. ROM is the Read Only Memory built into your IBM PC. The ROM contains a group of routines whose purpose is to operate the various peripherals connected to the computer, such as the video screen, keyboard, and disk drives. These routines are called collectively ROM BIOS, for "Basic Input/Output System."

This part of ROM occupies the absolute memory addresses from FE000h up to the top of memory at FFFFFh, which is 2000h bytes, or 8K of memory. Another, larger part of ROM is dedicated to the BASIC

interpreter. This starts at F6000 and goes up to the bottom of the ROM BIOS, which is 32K. We aren't going to be concerned with ROM BASIC, the unraveling of which would require another book in itself. In fact, IBM chose not to document the ROM BASIC, so there are no routines in it that we can use conveniently. (That doesn't mean there aren't all sorts of good routines in there if you wanted to disassemble the code with "U" and figure out what it does, but that's another story.) The location of the two kinds of ROM in relation to the entire 1-megabyte address space of the IBM PC is shown in Figure 9-1.

One thing to keep distinct is the difference between ROM BIOS and DOS. The ROM BIOS was installed in your computer by IBM, and is an integral part of the PC, like the case and the power cord. PC-DOS, on the other hand, is manufactured by Microsoft, Inc., and is not an integral part of the computer. In fact, it is even possible to run your PC with operating systems other than PC-DOS. Thus, although they have learned to work together very well, ROM BIOS and DOS are quite distinct and

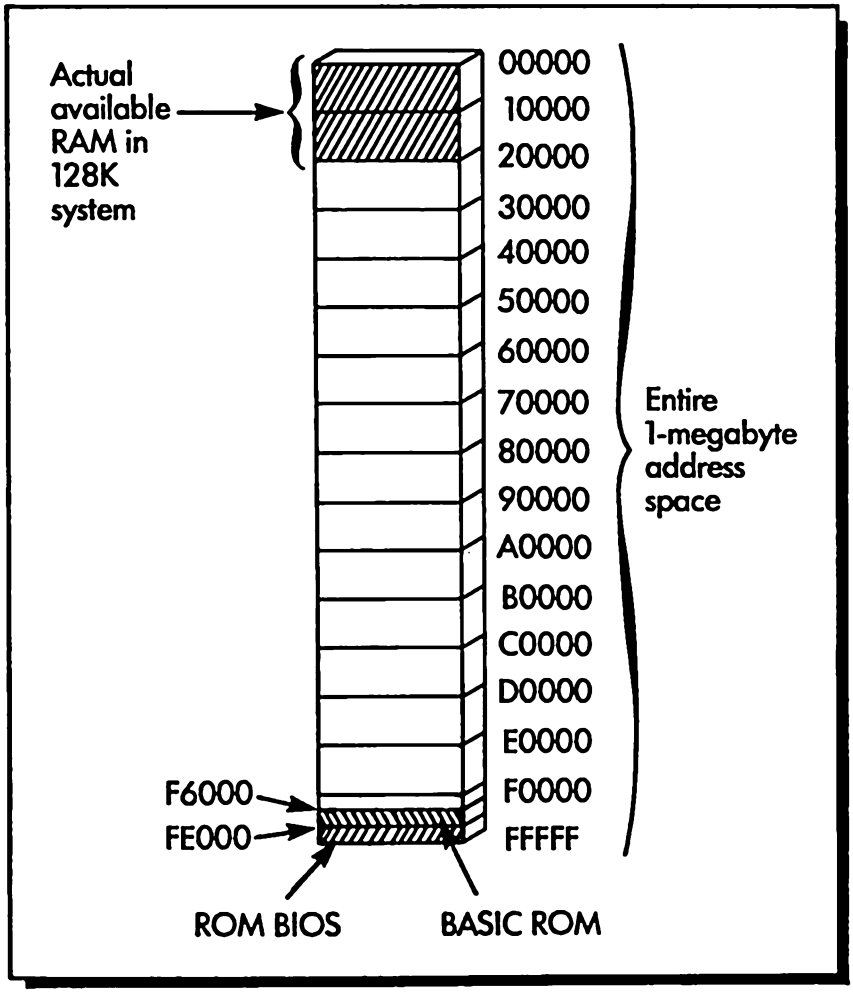


Figure 9-1. ROM location in address space

were implemented by different companies with somewhat different philosophies.

The ROM BIOS routines are — in a sense — on a lower level than the DOS functions (like "Display Output" and "Keyboard Input") that we've used in earlier chapters. That is, the ROM routines are closer to the hardware, and further away from our program. Generally, our program will call a DOS function, which will in turn call a ROM routine, which will then perform a specific I/O function such as printing a character on the screen.

ROM versus DOS

However, it sometimes happens that there is no DOS function that does what we want to do. In that case, our program can access the ROM routine directly, without going through DOS at all. There are advantages and disadvantages to this.

Generally (as we'll see) ROM routines are more flexible and versatile than DOS functions in dealing with a specific piece of hardware. On the other hand, this versatility can make them somewhat less convenient to use.

A more serious drawback to using ROM routines is that they can make your program less portable. What does that mean? Portability refers to the ability of a program to run on a variety of different machines. If you stick to using DOS functions, your program will probably work on any computer that can run a similar version of PC-DOS such as MS-DOS. However, if you use routines in the ROM BIOS, then your program must operate on a computer which has equivalent routines actually built into its own ROM. Building an IBM-equivalent ROM like this is a somewhat taller order for a manufacturer, so there are some so-called "IBM-compatible" PCs that will not run programs that make use of the ROM routines.

For programs with the maximum degree of portability, therefore, stick to DOS functions. However, since there are many things you can't do with DOS, we will show you some features that are only available in ROM, and explain how to use them. Learning about ROM will also prepare you for the next chapter, on graphics. There are, as we will learn, graphics routines built into ROM which greatly simplify drawing pictures and doing a variety of other things on the screen.

The ROM BIOS Listing

The first thing you need to know about the ROM BIOS is where to learn more about it. IBM has included a complete listing of the entire

ROM BIOS in appendix A of the *IBM Personal Computer Technical Reference* manual. This is rather amazing: many manufacturers are prone to a paranoid secrecy about their ROM. The listing is so packed full of routines, comments, and clever ideas that several books could be written about it alone. You should glance through it to get an idea of what's involved.

Interrupts and ROM Routines

Before we plunge into our first ROM routine, you should understand how your program accesses the ROM BIOS. Actually, you have already made use of this process for two different functions: first, to call DOS with an INT 21 instruction; and second, to terminate a program with an INT 20. What happens when these INT instructions are executed?

INT causes a software-generated interrupt to take place. (There are also hardware-generated interrupts, but we aren't going to discuss them in this book.) What's a software-generated interrupt? That's a somewhat complex story.

The first 1024d bytes in your PC's memory are used to hold a series of 256d addresses. These addresses are called "interrupt vectors." The word "vector" in this context means that the address is used to hold *another* address: a pointer to another location elsewhere in memory. Each vector has four bytes: two for the offset address of a particular routine, and two for its segment address. Each of these vectors is assigned to an interrupt number, which for some reason is called an *interrupt type*. The 256 interrupt types are numbered from 0 to FFh.

Some of these interrupt types are used by DOS, some are used by ROM BIOS, and some are even used by the BASIC interpreter. Interrupt type 21h is used by DOS for function calls, as you already know. The types we'll be concerned with here are the ones used for ROM routines. These are numbered from 10h to 17h. A complete list of interrupt types can be found in the section on ROM and System Usage in the *IBM Personal Computer Technical Reference* manual.

The interrupt vectors are used by the INT instruction to transfer control to a particular routine. Here's how the process works. Suppose you execute an INT 16h. This is essentially the same as a FAR CALL to the address in the interrupt vector number 16h. How can we tell what this address is? Let's get into DEBUG and prowl around. First we'll dump the beginning of the interrupt vector table:

```
A>debug
-dØ:Ø
ØØØØ:ØØØØ  43 31 DB ØØ 3F Ø1 7Ø ØØ-C3 E2 ØØ FØ 3F Ø1 7Ø ØØ   C1[.?.p.Cb.p?.p.
ØØØØ:ØØ1Ø  3F Ø1 7Ø ØØ 54 FF ØØ FØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ?.p.T..p........
ØØØØ:ØØ2Ø  A5 FE ØØ FØ 87 E9 ØØ FØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   %~.p.i.p........
ØØØØ:ØØ3Ø  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-57 EF ØØ FØ 3F Ø1 7Ø ØØ   ........Wo.p?.p.
ØØØØ:ØØ4Ø  65 FØ ØØ FØ 4D F8 ØØ FØ-41 F8 ØØ FØ 59 EC ØØ FØ   ep.pMx.pAx.pYl.p
ØØØØ:ØØ5Ø  39 E7 ØØ FØ 59 F8 ØØ FØ-2E E8 ØØ FØ D2 EF ØØ FØ   9g.pYx.p.h.pRo.p
ØØØØ:ØØ6Ø  ØØ ØØ ØØ F6 F2 E6 ØØ FØ-6E FE ØØ FØ 38 Ø1 7Ø ØØ   ...vrf.pn~.p8.p.
ØØØØ:ØØ7Ø  53 FF ØØ FØ A4 FØ ØØ FØ-22 Ø5 ØØ ØØ ØØ ØØ ØØ ØØ   S..p$p.p".......
-d
ØØØØ:ØØ8Ø  FB ØB DB ØØ 8Ø Ø1 34 Ø5-42 Ø2 ØØ Ø6 7Ø Ø2 ØØ Ø6   {.[...4.B...p...
ØØØØ:ØØ9Ø  E2 Ø4 34 Ø5 D4 14 DB ØØ-21 15 DB ØØ E7 27 DB ØØ   b.4.T.[.!.[.g'[.
ØØØØ:ØØAØ  Ø7 ØC DB ØØ 26 Ø1 7Ø ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ..[.&.p.........
ØØØØ:ØØBØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-6D Ø3 34 Ø5 ØØ ØØ ØØ ØØ   ........m.4.....
ØØØØ:ØØCØ  EA Ø8 ØC DB ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   j..[............
ØØØØ:ØØDØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
ØØØØ:ØØEØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
ØØØØ:ØØFØ  ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
```

There are four bytes per interrupt vector, so to find the address of a particular interrupt type, we multiply it by four. 16h is 22d, times 4 is 88d, which is 58h. Here are the four bytes at that address:

```
ØØØØ:ØØ5Ø  39 E7 ØØ FØ 59 F8 ØØ FØ-2E E8 ØØ FØ D2 EF ØØ FØ   9g.pYx.p.h.pRo.p
                                   |
```

Interrupt type 16h Address F000:E82E

As you know, 16-bit numbers are stored in memory with the least significant byte first. Similarly, the offset part of an address is stored before the segment address. So the address represented by the bytes shown above is F000:E82E. This is the same as absolute address FE82E, and that's the address of the keyboard routine in ROM, a fact we can verify by looking in the table of contents that precedes the ROM BIOS listing in the *IBM Personal Computer Technical Reference* manual.

Something else to notice in the dump above is that not all the possible interrupt addresses are used: many simply contain zeros.

The INT Instruction

Executing an INT 16h instruction will take us to address F000:E82E, which is the Keyboard I/O routine in the ROM BIOS. What do we find there? Look up the appropriate page in your ROM BIOS listing. You will find a routine labeled "INT 16". The routine begins with a series of

comments which explain how this routine is to be used. If we enter the routine with AH = 0, the routine will return the next character typed on the keyboard in AL, and the scan code in AH. This sounds a good bit like the Keyboard Input DOS function, except for the new phrase "scan code." What does this mean?

INT Instruction

Calls an interrupt routine.

Transfers control to the routine whose address is at n * 4, where n is the interrupt type number.

Also sets up return by placing flag register, segment address, and offset address of instruction following the INT (that is, contents of CS and IP registers) on the stack. Clears the trap flag and interrupt flags.

Return from routine must be via IRET instruction.

Flags affected: IF, TF

Scan Codes and the Keyboard

The ordinary letters on the keyboard, like "a" and "b", generate an ASCII code. You've already made use of this operation when you used the Keyboard Input DOS function call. But what happens when you press keys that don't have an ASCII code, like the function keys and the Caps Lock or Num Lock keys? In this case the DOS functions become somewhat awkward to use: you have to make the DOS call *twice:* the first time it returns a 00; the second time it returns a number which is (usually) something called the *scan code* (which we'll describe soon). The ROM routine we are about to describe, on the other hand, returns both the ASCII code (if one exists), and the scan code at the same time.

Every key has a Scan Code (except the shift keys like ⇧, Ctrl, Caps Lock, and Alt), but not every key has an ASCII code.

Let's write a program that will use interrupt type 16h to call the Keyboard I/O ROM routine, and print out the numbers returned by the routine. Since the Scan Code is returned in AH and the ASCII code in

AL, we can see *both* these aspects of each key press simply by printing out the contents of AX. To do this we'll combine the BINIHEX routine we wrote earlier to print out the contents of a register in hex, with some very simple new code to call the ROM routine. We'll also print the actual character (if it exists in printable form). This requires a separate step — a DOS function call — because the ROM routine does not automatically echo the key press to the screen. The program then types a carriage return and linefeed, so it will be ready for the next character on the next line.

Keyboard Input/Output Program

```
                        ;KEYBOARD I/O TEST--Prints out
                        ;scan code and ASCII of any key

= ØØØ2                  display equ     2h      ;display character fnc
= ØØ21                  doscall equ     21h     ;DOS interrupt routine

                        ;*********************************************

ØØØØ                    pro_nam segment         ;define code segment

                        ;---------------------------------------------
ØØØØ                    main    proc    far     ;main part of program

                                assume  cs:pro_nam

ØØØØ  B4 ØØ             again:  mov     ah,Ø    ;read character funct
ØØØ2  CD 16                     int     16h     ;keyboard I/O ROM call
ØØØ4  8B D8                     mov     bx,ax   ;move AX to BX
ØØØ6  E8 ØØ25 R                 call    binihex ;print scancode & char
ØØØ9  B2 2Ø                     mov     dl,2Øh  ;print space
ØØØB  B4 Ø2                     mov     ah,display
ØØØD  CD 21                     int     doscall
ØØØF  8A D3                     mov     dl,bl   ;print character
ØØ11  B4 Ø2                     mov     ah,display ;  in ASCII
ØØ13  CD 21                     int     doscall
ØØ15  B2 ØD                     mov     dl,Ødh  ;print return
ØØ17  B4 Ø2                     mov     ah,display
ØØ19  CD 21                     int     doscall
ØØ1B  B2 ØA                     mov     dl,Øah  ;print linefeed
ØØ1D  B4 Ø2                     mov     ah,display
ØØ1F  CD 21                     int     doscall

ØØ21  EB DD                     jmp     again   ;get another one

ØØ23  CD 2Ø                     int 2Øh ;return from program to DOS
```

```
ØØ25                            main    endp    ;end of main part of program
                                ;-----------------------------------------
ØØ25                            binihex proc    near

                                ;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
                                ;  TO HEX ON CONSOLE SCREEN

ØØ25  B5 Ø4                             mov     ch,4    ;number of digits
ØØ27  B1 Ø4                     rotate: mov     cl,4    ;set count to 4 bits
ØØ29  D3 C3                             rol     bx,cl   ;left digit to right
ØØ2B  8A C3                             mov     al,bl   ;move to DL
ØØ2D  24 ØF                             and     al,Øfh  ;mask off left digit
ØØ2F  Ø4 3Ø                             add     al,3Øh  ;convert hex to ASCII
ØØ31  3C 3A                             cmp     al,3ah  ;is it > 9 ?
ØØ33  7C Ø2                             jl      printit ;no, so Ø to 9 digit
ØØ35  Ø4 Ø7                             add     al,7h   ;yes, so A to F digit

ØØ37                            printit:
ØØ37  8A DØ                             mov     dl,al   ;put ASCII char in DL
ØØ39  B4 Ø2                             mov  ah,display ;display output funct.
ØØ3B  CD 21                             int     doscall ;call DOS
ØØ3D  FE CD                             dec     ch      ;done 4 digits?
ØØ3F  75 E6                             jnz     rotate  ;not yet
ØØ41  C3                                ret             ;done subroutine

ØØ42                            binihex endp
                                ;-----------------------------------------
ØØ42                            pro_nam ends    ;end of code segment
                                ;*****************************************

                                        end     ;end assembly
```

As you can see, the real meat of this program is in just four lines:

```
mov   ah,Ø      ;read character function
int   16h       ;keyboard I/O ROM call
mov   bx,ax     ;move AX to BX
call  binihex   ;print scancode & char
```

We set up for this interrupt by putting a function number in the AH register (0 in this case), just as we do for function calls. The scan code and ASCII codes are returned in AH and AL, so we print them out together with BINIHEX.

What do the scan codes look like? Assemble the program as an EXE file and run it. Press some keys. Here's the result for some lowercase letters:

```
1E61    ←— a
3Ø62    ←— b
2E63    ←— c
2Ø64    ←— d
└┘└┘
 └┘└──ASCII codes
   └────Scan codes
```

As you can see, the scan codes aren't related to the ASCII codes, or to each other, in any easily understood way. It seems that characters close together on the keyboard tend to have scan codes that are close together numerically. Here's what the function keys produce:

```
3BØØ    ←— F1
3CØØ    ←— F2
3DØØ    ←— F3
3EØØ    ←— F4
3FØØ    ←— F5
4ØØØ    ←— F6
41ØØ    ←— F7
42ØØ    ←— F8
43ØØ    ←— F9
44ØØ    ←— F1Ø
```

The keys on the numeric keypad also have scan codes. These codes will be different if the Alt, Ctrl, or ⇧ keys are depressed at the same time. Try it!

Now you know how to find out about all the keyboard keys. This is invaluable for writing games and for any program that uses the function keys or cursor control keys. A word-processing program is an obvious example of a serious program that would make use of the scan codes for this purpose.

The IRET Instruction

You won't need to use an IRET instruction in your program, unless you're writing your own interrupt routine. However, we'll show you what it looks like. IRET is simply the way the routine which is called with an INT instruction returns to the calling program. IRET reverses what INT did, POPping the appropriate addresses off the stack, and restoring them to the IP and CS register. It's very much like a FAR RET following a CALL to a FAR PROCedure.

IRET Instruction

Returns from interrupt to calling program.

POPs contents of IP register, CS register, and flag register off the stack, thus restoring control to instruction following INT in calling program.

Flags affected: all (but previous values are restored)

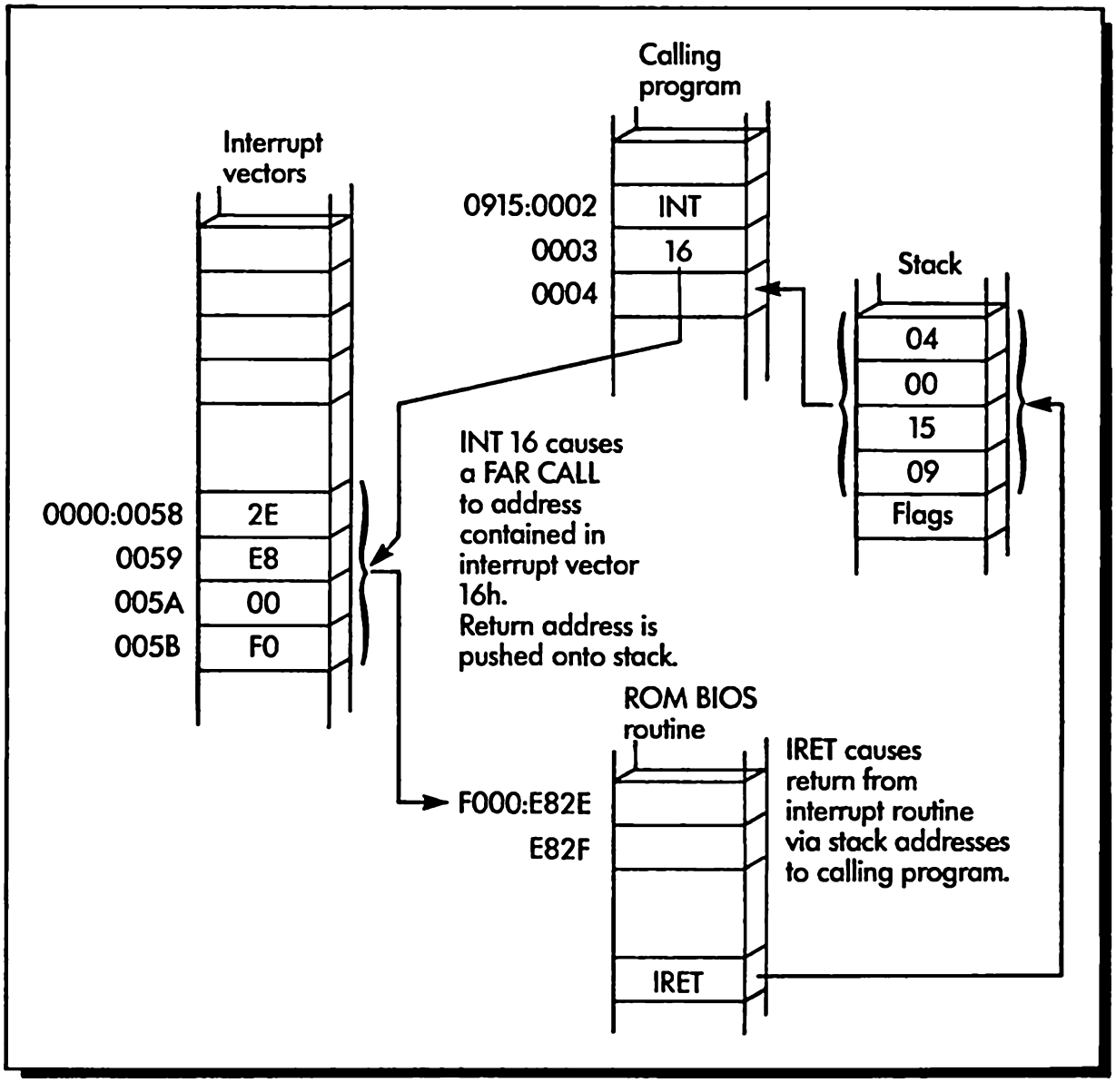


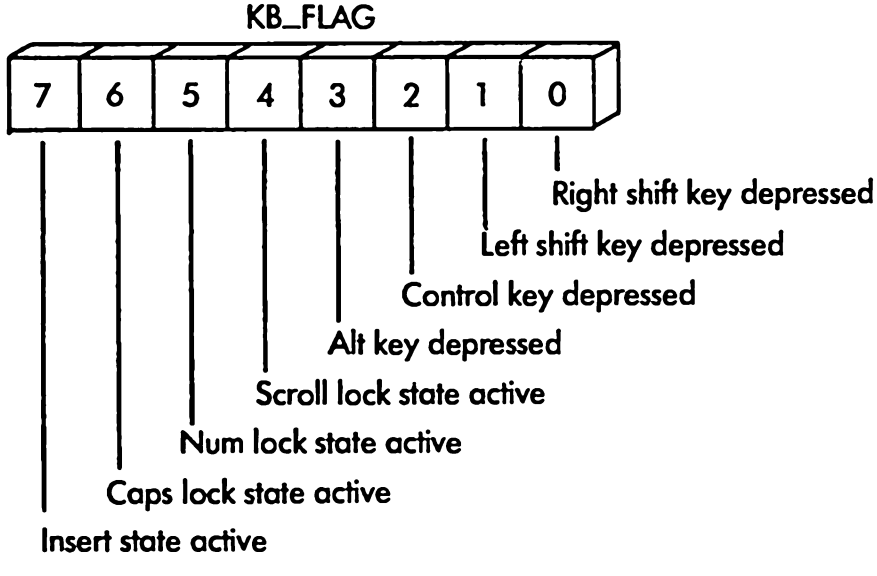
Figure 9-2. Operation of the INT and IRET instructions

Figure 9-2 summarizes how control is transferred from the calling program to the interrupt routine in the ROM BIOS, and back again.

Finding the Shift Status

Even though you can read the scan codes, you still don't know everything about the keyboard. The ⇧, Ctrl, Alt, Num Lock, Scroll Lock, Ins, and Caps Lock keys don't have scan codes. This is because they do not generate characters of their own: they only *change* the codes generated by other keys. So how can your program find out whether, for instance, the Num Lock key is depressed, or whether it has been toggled to read numbers or cursor control keys from the numeric keypad?

To answer this question we must delve further into the mysterious world of the ROM BIOS listing in appendix A of the *IBM Personal Computer Technical Reference* manual. You'll note in the comments at the start of the INT 16 section of the listing that there is a function which will return the current "shift status" if register AH contains 2. The bit settings for the shift status code can be found in the "equates" for a variable called "KB_FLAG". Equates are just EQU pseudo-ops used to define various parameters at the beginning of the program. The EQUs which follow after "shift flag equates within KB_FLAG" tell us that the bits of this flag are used as shown in the following figure:



Thus by using this function of the INT 16h keyboard I/O routine, we can read a byte which tells us all about the shift keys and the shift states on the keyboard. Here's a short program to do it. This program reads the KB_FLAG continuously, so that whenever a shift key is depressed, this fact shows up immediately on the screen. (Be careful, though — if a shift

key stays depressed for too long, you may have to cheer it up with a few kind words or a bouquet of roses.)

```
                        ;SHIFT STATUS TEST PROGRAM--
                        ;  Displays shift status continuously

= 0002                  sh_stat equ     2h      ;shift status function
= 0016                  key_rom equ     16h     ;keyboard ROM call

= 0002                  display equ     2h      ;DOS display routine
= 0021                  doscall equ     21h     ;DOS interrupt number

= 000D                  return  equ     0dh     ;carriage return

                        ;*****************************************
0000                    prognam segment ;define code segment

                        ;-----------------------------------------
0000                    main    proc    far     ;main part of program

                                assume  cs:prognam

0000  B4 02             again:  mov  ah,sh_stat ;sh status function
0002  CD 16                     int  key_rom    ;call kbd ROM routine
0004  8B D8                     mov  bx,ax      ;put result in BX
0006  E8 0011 R                 call binihex    ;print out result
0009  B2 0D                     mov  dl,return  ;print carriage return
000B  B4 02                     mov  ah,display
000D  CD 21                     int  doscall
000F  EB EF                     jmp  again      ;repeat

0011                    main    endp    ;end of main part of program
                        ;-----------------------------------------
0011                    binihex proc    near

                        ;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
                        ;  TO HEX ON CONSOLE SCREEN

0011  B5 04                     mov     ch,4    ;number of digits
0013  B1 04             rotate: mov     cl,4    ;set count to 4 bits
0015  D3 C3                     rol     bx,cl   ;left digit to right
0017  8A C3                     mov     al,bl   ;move to DL
0019  24 0F                     and     al,0fh  ;mask off left digit
001B  04 30                     add     al,30h  ;convert hex to ASCII
001D  3C 3A                     cmp     al,3ah  ;is it > 9 ?
001F  7C 02                     jl      printit ;no, so 0 to 9 digit
0021  04 07                     add     al,7h   ;yes, so A to F digit
0023                    printit:
0023  8A D0                     mov     dl,al   ;put ASCII char in DL
```

```
0025  B4 02                   mov  ah,display ;display output funct.
0027  CD 21                   int     doscall ;call DOS
0029  FE CD                   dec     ch      ;done 4 digits?
002B  75 E6                   jnz     rotate  ;not yet
002D  C3                      ret             ;done subroutine

002E                    binihex endp
                        ;-----------------------------------------
002E                    prognam ends    ;end of code segment
                        ;*****************************************

                              end       ;end assembly
```

Type in the program and give it a try. You'll learn everything you ever wanted to know about the keyboard shift status.

This has been a brief introduction to the keyboard ROM routines. In the next section we'll explore another part of ROM: the routines used to send information to the video screen.

Video ROM Routines

The video ROM routines are considerably more complicated and extensive than the keyboard routines. In addition to manipulating the monochrome screen, they can also be used for plotting points on the color screen. We'll be covering their use with color graphics in the next chapter. For the moment, let's learn some useful ROM routines on the monochrome display.

IBM's ROM BIOS listing (appendix A of the *IBM Personal Computer Technical Reference* manual) has many pages of comments describing how to use the Display (VIDEO) I/O interrupt routine. A tip of the hat to IBM for providing this information.

One of the things it's nice to be able to do from your assembly language program is to clear the screen. In BASIC and DOS 2.00 you do this by executing a CLS instruction, but there is no such instruction in 8088 code. However, a ROM routine makes it fairly easy.

Check the ROM BIOS listing for a description of the function called Scroll Active Page Up. This routine is invoked by using an INT 10h instruction with the AH register holding a 6. We'll see how to use this routine more completely in a minute; for the moment we're only interested in how to use it to clear the screen. The comments tell us that AL=0 will "blank the entire window." What does this mean?

The "window" is simply a section of the screen, described as being so many rows high and so many columns wide. The upper left-hand corner of the screen is the starting place for these measurements, since it's here

that both the column and row numbers are zero. The row numbers run down the screen from 0 to 24, and the column numbers run across the screen from 0 to 79 (since it's a 25 by 80 screen). Figure 9-3 shows how this looks.

The Scroll Up function needs coordinates for two corners of a rectangular window: the upper left-hand corner, and the lower right-hand corner. These coordinates must be placed in the following registers:

```
CH    <— Upper left row
CL    <— Upper left column
DH    <— Lower right row
DL    <— Lower right column
```

Using this routine to clear the screen consists of putting these values into the registers for that portion of the screen, or "window," that we

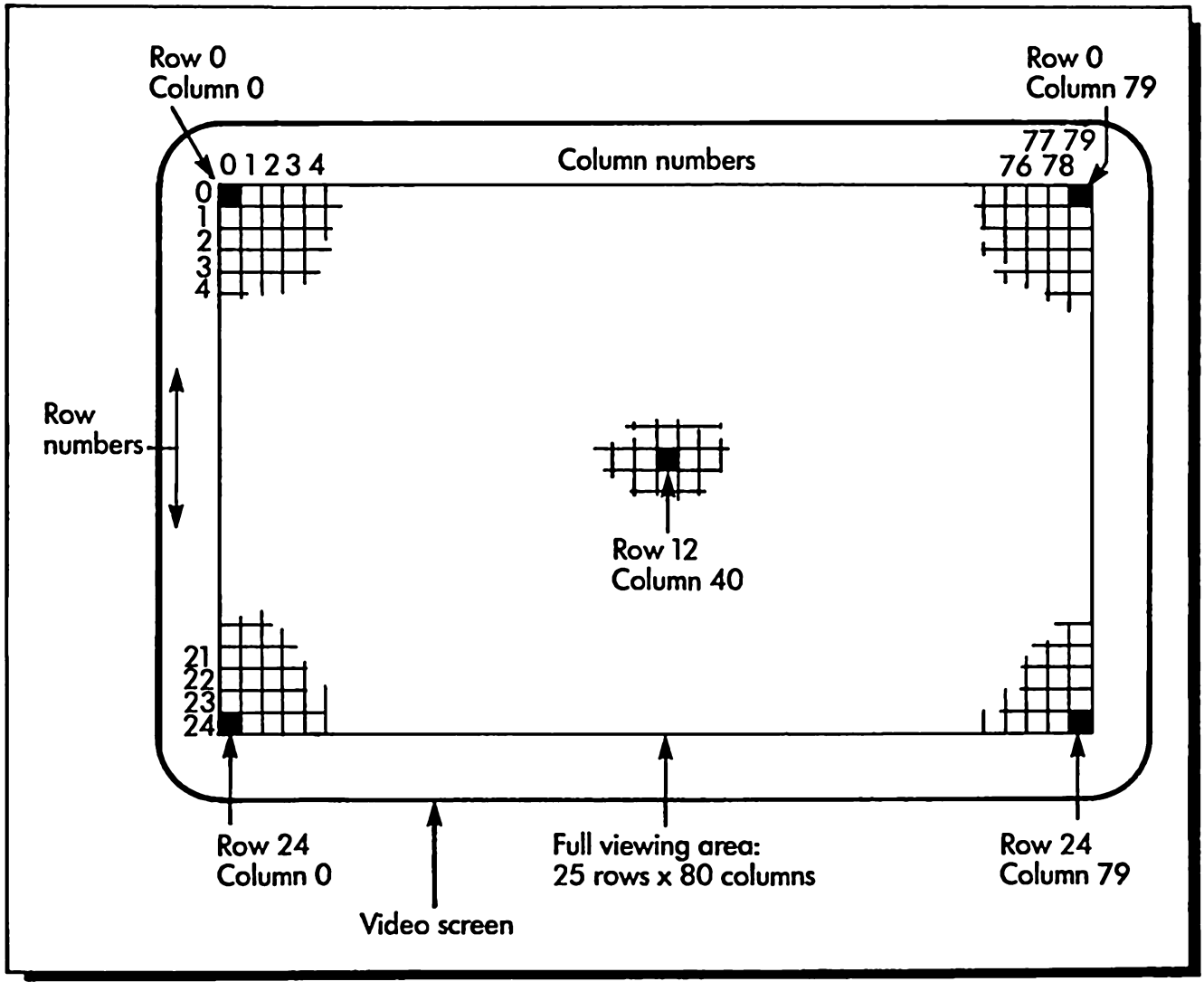


Figure 9-3. Character positions on the monochrome screen

want to clear. AH must hold a 6 to designate the Scroll Up function, and AL must be 0, to tell the function that we want to clear the screen. For example, if we want to clear the *entire* screen, the following code fragment will do the trick:

```
;clear screen, using scroll up function

mov  ah,6       ;scroll up function
mov  al,Ø       ;code to blank screen
mov  ch,Ø       ;upper left row
mov  cl,Ø       ;upper left column
mov  dh,24      ;lower right row
mov  dl,79      ;lower right column
mov  bh,7       ;blank line attribute
int  1Øh        ;video ROM call
```

Attributes

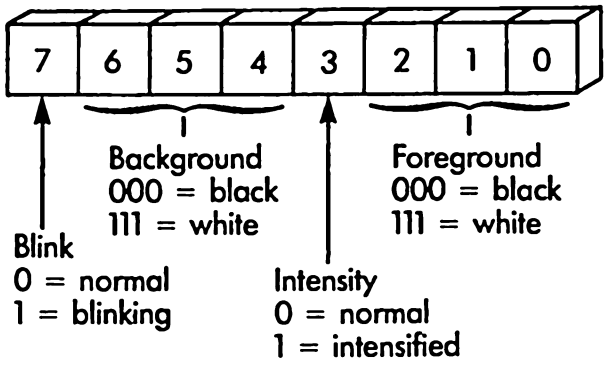
One point which we didn't mention, but which you can see in the above listing, is that the BH register must be set to a particular *attribute* number. What does this mean?

Each of the 2000d (25 times 80) character positions on the monochrome display is actually represented by *two bytes*. The first of these bytes is simply the ASCII code for the character at that position. The second byte is something called an *attribute byte*. This attribute byte determines whether a character is blinking or not, whether it is intensified or not, and whether it is in "reverse video" (black on white) or not.

> Every character on the monochrome screen has its own attribute: blinking or not, intensified or not, normal or reverse video.

A single byte is used to hold the attribute number for each character. We can give a particular character an attribute by using the Scroll Up function. When we clear the screen with this function, all we have to do is change the value in BH when we call the function. In the code above, since the entire screen is cleared, *all* the characters on the screen will be given the particular attribute.

The individual bits in the attribute byte are arranged as shown below:



How do you use this information? Suppose you want want to write characters on the screen in reverse video; that is, black figures on white, instead of the normal white on black. The "normal" attribute is 07h, which in binary is 00000111. As you can see in the figure above, that means the foreground will be white, the background will be black, and the blink and intensity bits will be normal. To change to reverse video we would use an attribute of 70h, which makes the foreground black and the background white.

If we wanted white on black, but blinking, we'd use 87h (10000111 binary). The attributes can be combined in any combination. So for instance, we could have reverse intensified blinking by using F1h (11110001 binary).

In the next section we'll show you how to change the attributes so you can see these effects in action, but in the program we're putting together now we want normal video characters, so the number we put into the BH register is 7. The "clear screen" code fragment shown above will cause the entire screen to be cleared. We aren't going to make this into an executable program; instead, we're going to combine it with some other routines into a larger program in the next section.

Windows

The ROM routines in the IBM PC give you the ability to do something in assembly language that few other personal computers can: divide the screen into different areas called "windows," which can be used independently of each other. This feature is used by many advanced programs, especially the new breed of "integrated" software. These programs combine the abilities of two or more traditional programs, such as word processors, databases, and spreadsheets, into a single program.

How would such a program use windows? Suppose you were writing a letter on the word-processor part of the program, and you wanted to include a list of sales figures which were obtainable in the database program. You could keep the letter in one window on one side of the

screen, and look through the database in a separate window on the other side of the screen for the figures you wanted. When you found them, you could insert them into your letter, a process greatly facilitated by being able to see both your letter and the data at the same time.

It's beyond the scope of this book to present such a program. What we will do is show you how to get a single window to work on your screen. When you need several windows, you can then use one in each routine. As many windows as you want can coexist on the screen at the same time (providing they all fit, of course).

The following program creates a window 20 columns across and 9 rows high in the middle of your screen. It then waits for you to type something. Whatever you type will be displayed in this window. The characters will be entered on the bottom line of the window, and when each 20-character line is full it will scroll upward. Lines at the top of the 9-line high window will scroll into oblivion, never to be seen again, just as lines do at the top of the normal display screen. Figure 9-4 shows what the window looks like.

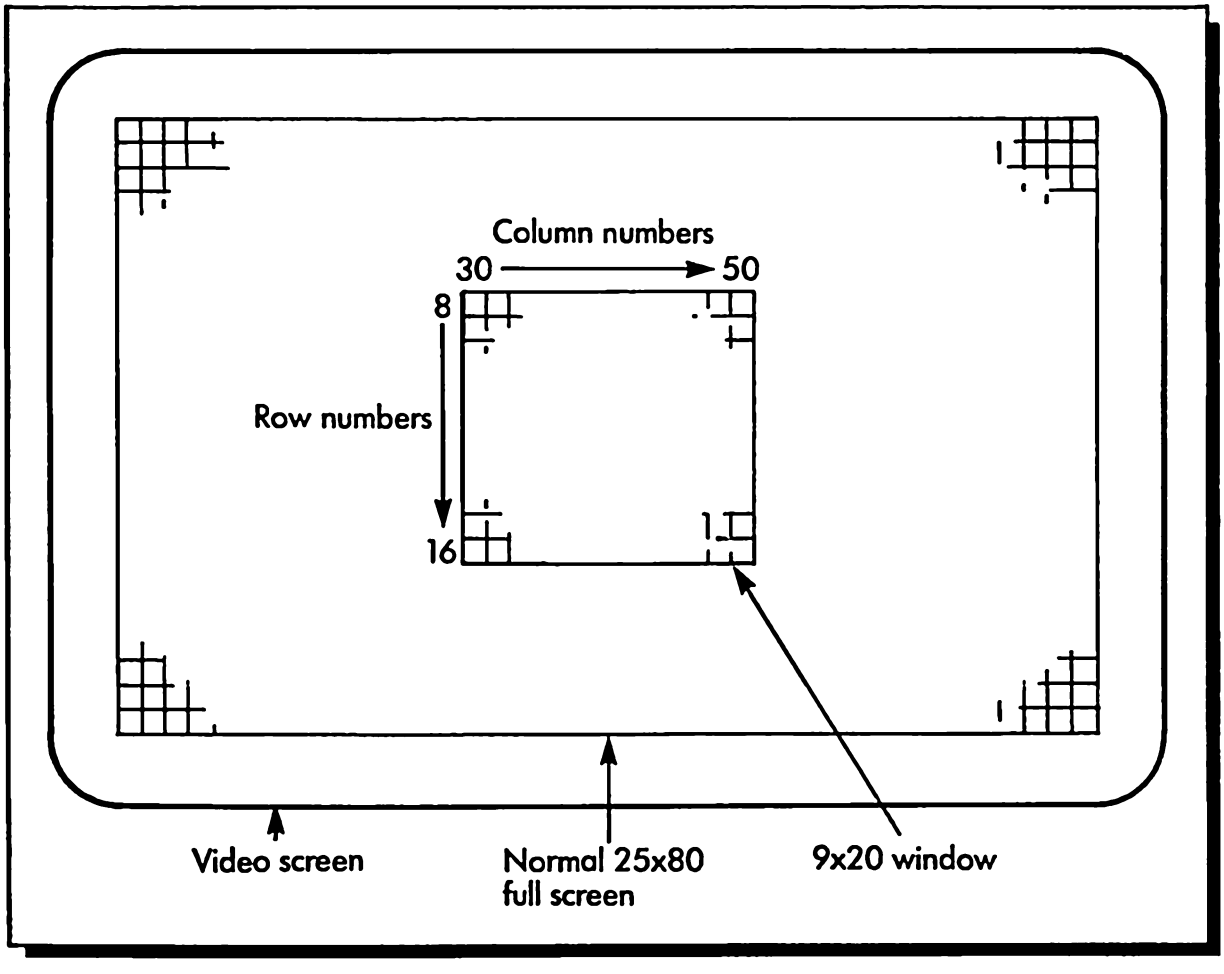


Figure 9-4. Small window in center of screen

Here's the listing for the complete WINDOW program:

```
                        ;WINDOW--Demonstrates video window function
                        ;  Uses ROM routines

                        ;keyboard writes into a window 2Ø chars wide
                        ;  and 9 chars high in middle of screen

                        ;*********************************************

ØØØØ                    prognam segment         ;define code segment

                                assume  cs:prognam

                        ;clear screen, using scroll up function

ØØØØ  B4 Ø6                    mov  ah,6       ;scroll up function
ØØØ2  BØ ØØ                    mov  al,Ø       ;code to blank screen
ØØØ4  B5 ØØ                    mov  ch,Ø       ;upper left row
ØØØ6  B1 ØØ                    mov  cl,Ø       ;upper left column
ØØØ8  B6 18                    mov  dh,24      ;lower right row
ØØØA  B2 4F                    mov  dl,79      ;lower right column
ØØØC  B7 Ø2                    mov  bh,7       ;blank line attribute
ØØØE  CD 1Ø                    int  1Øh        ;video ROM call

                        ;position cursor at bottom of window
ØØ1Ø                    pos_curse:

ØØ1Ø  B4 Ø2                    mov  ah,2       ;position cursor funct
ØØ12  B6 1Ø                    mov  dh,16      ;starting row
ØØ14  B2 1E                    mov  dl,3Ø      ;starting column
ØØ16  B7 ØØ                    mov  bh,Ø       ;current page
ØØ18  CD 1Ø                    int  1Øh        ;video ROM call

                        ;get characters from keyboard

ØØ1A  B9 ØØ14                  mov  cx,2Ød     ;set count to 2Ø

ØØ1D                    get_char:
ØØ1D  B4 Ø1                    mov  ah,1       ;kbd input function
ØØ1F  CD 21                    int  21h        ;call DOS
Ø021  3C Ø3                    cmp  al,3       ;if char is ctrl-C
Ø023  74 14                    jz   exit       ;  then exit
Ø025  E2 F6                    loop get_char

                        ;scroll up
Ø027  B4 Ø6                    mov  ah,6       ;scroll up function
Ø029  BØ Ø1                    mov  al,1       ;number of lines
Ø02B  B5 Ø8                    mov  ch,8       ;upper left row
```

```
ØØ2D  B1 1E                         mov  cl,3Ø      ;upper left column
ØØ2F  B6 1Ø                         mov  dh,16      ;lower right row
ØØ31  B2 32                         mov  dl,5Ø      ;lower right column
ØØ33  B7 Ø2                         mov  bh,7       ;normal attribute
ØØ35  CD 1Ø                         int  1Øh        ;video ROM call

ØØ37  EB D7                         jmp  pos_curse  ;go reset cursor

ØØ39  CD 2Ø                 exit:   int  2Øh

ØØ3B                        prognam ends    ;end of code segment
                            ;*******************************************

                                    end     ;end assembly
```

As you can see, this program makes use of several different video ROM routines: Clear Screen, Position Cursor, and Scroll Up. The first thing we do is to clear the entire screen. If you had several windows operating on your screen at once, you would want to able to clear them separately. This would be easily accomplished by changing the row and column values in the Clear Screen function.

Next we want to position the cursor at the lower left-hand corner of the window, so that when we start to type, the letters will go in the right place. This is done with another video ROM function, Position Cursor, which is activated with an INT 10h and AH=2, as shown in the following code fragment:

```
;position cursor at bottom of window
        mov  ah,2       ;position cursor funct
        mov  dh,16      ;starting row
        mov  dl,3Ø      ;starting column
        mov  bh,Ø       ;current page
        int  1Øh        ;video ROM call
```

As you can see, all we have to do to activate this function is specify the row and column where we want the cursor to go, and the "current page." What does "page" mean? It turns out that it's possible to have different screens full of information sitting in memory at the same time, so you can switch back and forth from one to another. This makes it possible to to fill the screen almost instantly with an entirely different picture. We aren't going to explore this feature of the IBM PC in this book, so we simply set BH to 0, which means the current or "normal" page.

Now that the cursor is properly positioned, we use the old familiar DOS function Keyboard Input to get the characters from the keyboard and

put them on the screen. However, we don't want to exceed the 20-character width of the window, so we use a LOOP instruction to count 20d characters.

At the end of the line we activate the Scroll Up function. This is the same as Clear Screen, except that AL = 1, the number of lines to scroll up, rather than 0. Also, we put in the coordinates of our window, rather than the coordinates of the full screen. When we execute this function, all the lines in the window scroll up, and a new line appears at the bottom of the window with the attribute specified in BH. Again, we use the normal attribute 07h here.

Type in this program, assemble it, and try it out. It's kind of fun to see the screen reduced to mini-size, and to imagine all the wonderful things you can do with multiple windows.

Changing the Attribute

We promised you earlier that we'd show you how to change the attribute. Call up the WINDOW program shown above with DEBUG:

```
A>debug window.exe
```

Now what we want to do is modify the attribute byte in the Scroll Up function. The instruction that does that is on line 0033, and the attribute byte is the second byte in this instruction, at 0034, so that's what we'll change. Here's what that line looks like:

```
ØØ33  B7 Ø7                    mov  bh,7       ;normal attribute
          |
          Byte to be changed.
```

Let's make it reverse video, which is an attribute of 70h. Since this is an EXE file, the data segment and code segment have different segment addresses, so we need to type the segment address as well as the offset address when we use the "E" command to change the byte:

```
-e915:34 ———————— Type the segment and offset address
Ø915:ØØ34  Ø7.7Ø ———— Type 70, the attribute for reverse video
|
Segment address (may be different on your system)
```

To make sure you're at the beginning of the program before you run it, set IP back to 0000:

```
-rip          ← Make sure IP is 0
IP ØØØØ
:Ø
-g            ← Run the program again
```

When you start to type into the window everything will seem normal for the first line. But when the second line scrolls into view, it will be in reverse video. To really make this program look good you would want to make the first line reverse video as well. To do this you would write another section of the program to clear only the bottom line, setting BH to the reverse video attribute 70h.

Try it. After the first line has scrolled off the top of the window, you have an entire little rectangle of text in reverse video! If you had several different windows, you could give them all different attributes: one blinking, one reverse video, and so on. The possibilities are endless.

Summary

In this chapter you've learned how to use some of the routines built into your PC's Read Only Memory. There are other routines which we haven't covered; you can discover some of these for yourself by reading the ROM BIOS listing in your *IBM Personal Computer Technical Reference* manual. There are also a number of graphics-oriented video routines; we're going to investigate these in the next chapter.

10

Monochrome and Color Graphics

Concepts

Memory-mapped graphics
Plotting a dot — Monochrome
Changing graphics modes
Plotting a dot — Color
Drawing a line — Color

8088 Instructions

TEST = Test bits
LABEL = Defines variable type (pseudo-op)
SEGMENT AT = Specifies absolute segment address (pseudo-op)

ROM BIOS Functions

Set Graphics Mode
Write Dot

Applications

Drawing pictures — Monochrome
Drawing pictures — Color
Line-drawing routine

Writing graphics programs is one of the most exciting and rewarding fields in programming. The finished product is something everyone can relate to. While writing an extra-fast sort routine may elicit yawns from all but the knowledgeable few, a dazzling graphics program will win oohs and ahs even from children and grannies. Also, program development is simplified in graphics work because mistakes are so easy to find: The image on the screen looks wrong, and the way in which it

looks wrong points the way to the error in the program.

Graphics is a field in which assembly language shows itself to full advantage. Because an image on a video screen consists of thousands, or tens of thousands of elements, manipulating these images requires a great many programming instructions. The speed advantage that assembly language has over higher-level languages is critical here: Most advanced graphics techniques, such as animation, are possible only in assembly language.

In this chapter we're going to introduce you to how this fascinating visual world works in the IBM PC. We'll start off with a description of the monochrome screen. Although the graphics possibilities are limited here, it serves as an introduction to the more complicated color graphics modes, which will be covered next.

Graphics Modes in the IBM PC

In this section we're going to briefly describe the differences between the two main graphics modes on the IBM PC: monochrome and color.

Monochrome Display

So far we have assumed that you were using the monochrome monitor that comes with the IBM PC. It's a high-quality black and green monitor which is excellent for displaying text. However, it is less useful for displaying graphics — pictures — on the screen. The reason is that there are only 2000d separately-addressable positions on this monitor: 80 columns across by 25 rows down. Each of these 2000d locations is called a "pixel" in the graphics business. (IBM calls pixels "pels" for "picture elements.") This number of pixels provides very poor resolution for drawing pictures.

The effective resolution can be improved in certain situations by using the *graphics characters* which are built into the PC. Thus, if you want to draw a box, there are characters that will print on the screen as lines and corners; so that by combining them cleverly you can achieve a box with much finer lines than you could by simply turning on some of the 2000 pixels in a box-like pattern. (For a detailed discussion of character graphics on the monochrome screen, see *BASIC Primer for the IBM PC and XT*, by Bernd Enders and Bob Petersen [New York: Plume/Waite, New American Library, 1984].) Figure 10-1 shows the difference between using special characters to draw a box, and simply turning "on" some of the pixels to do the same job.

However, drawing boxes is a special case: in general there's no point

in trying to draw sophisticated pictures on the monochrome screen.

Color Display

Fortunately IBM has provided for a different kind of video display: the "color board," an option which permits you to use a color monitor or standard color TV set with your PC. The color board is a printed circuit board (a "card") that plugs into one of the card slots in your machine. It contains the circuitry that lets the computer communicate with a TV set or monitor. To use a TV set you will need not only the color card, but an RF (radio frequency) modulator to turn the signals from the color card into a form similar to TV broadcast signals that is digestible by your TV set. If you have an RGB (red, green, blue) color monitor you won't need this modulator. Also, you'll get better color and finer resolution than a TV set can deliver.

There are actually two different ways to use this color capability. Depending on which mode you prefer, you can either have a picture with 320 pixels horizontally and 200 vertically, where each pixel can be any of four colors; or you can have a picture with 640 pixels horizontally and 200 vertically, but in black and white. Either of these options can be selected by software (assuming the color card is installed). We'll show you how to do this later in the chapter.

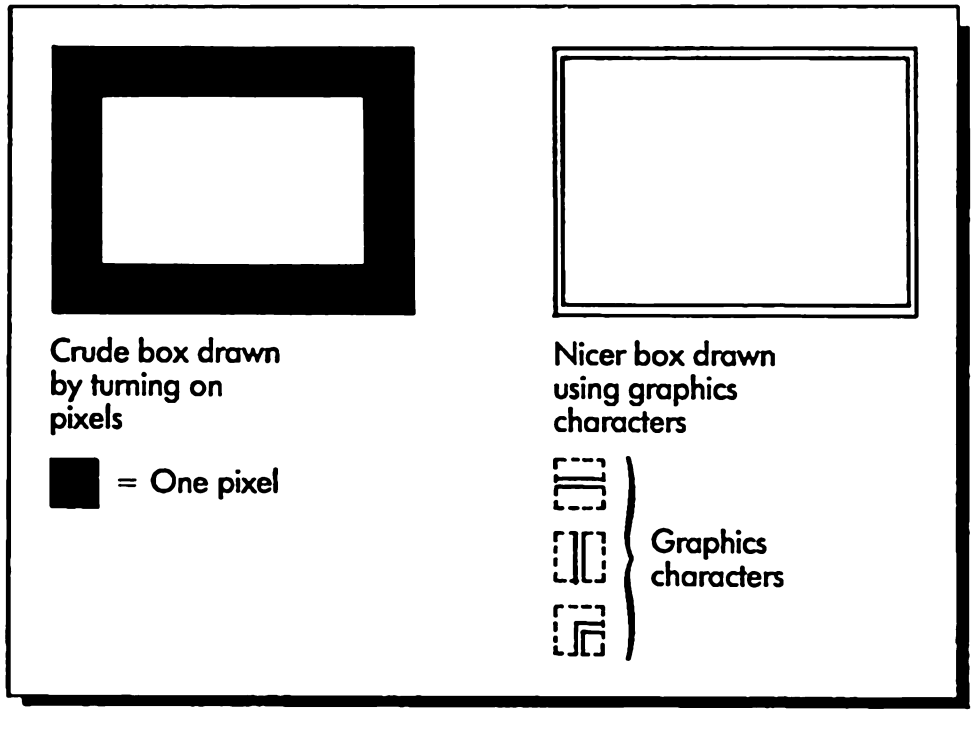


Figure 10-1. Boxes drawn with special characters on monochrome screen

Memory-Mapped Graphics

The IBM PC uses *memory-mapped* graphics. What does this mean? One of the questions the designers of a computer system have to answer is this: How will the information which is to be displayed be transmitted from the computer to the screen?

One solution is simply to send one character at a time to the screen, like a stream of parts on a conveyer belt. That's the effect we get when we use the Display Output DOS function: We send a character, it gets displayed on the screen at the current cursor position, and that's it. With this system we don't worry *where* the character goes on the screen: we assume it will simply be displayed following the last one we sent. This system has been in use in many operating systems, such as CP/M, for a long time. It works quite well if what you want to put on the screen is confined to ASCII characters or simple graphics characters.

> In memory-mapped graphics there is one location in memory corresponding to each pixel on the screen.

However, if you want to draw real pictures on the screen, you need a more sophisticated method. The IBM PC uses memory-mapped graphics. This means that for every pixel on the video screen, there is a corresponding location in memory which tells the video circuitry what to put on the screen in that location. Figure 10-2 shows what we mean by this.

Memory Mapping in the Monochrome Display

What does memory mapping mean in the case of the monochrome display? Imagine that you type the name "Euclid" on the keyboard, so that these letters appear on the screen. There they are in green, right before your eyes. Now, in your computer's memory there is a fixed address corresponding to the location on the screen of each of the letters E,u,c,l,i,d. If you knew where to look, you would, in fact, see the ASCII code for the letters. Want to try it?

Hop into DEBUG, and when you get the prompt simply enter the name "Euclid". You'll get an error, because of course DEBUG is not familiar with the famous founder of geometry.

```
A>debug
-Euclid
  'Error
```

Next you want to arrange things so the name "Euclid" is on the *second line from the top* of your screen (you'll see why in a minute). So just keep pressing ↵ until it gets up there:

```
-                ←— One dash (DEBUG prompt)
-Euclid          ←— The name is one line from the top of the screen
  'Error
-                ←— Dashes from pressing ↵
-
-
```

Now for the payoff. We're going to use the "D" command to dump the part of memory that contains the characters in the name "Euclid". So we type "d" followed by the segment address of the monochrome video memory — which is B000h — then a colon, then the offset address — which is 0000 for the character in the upper left-hand corner of the

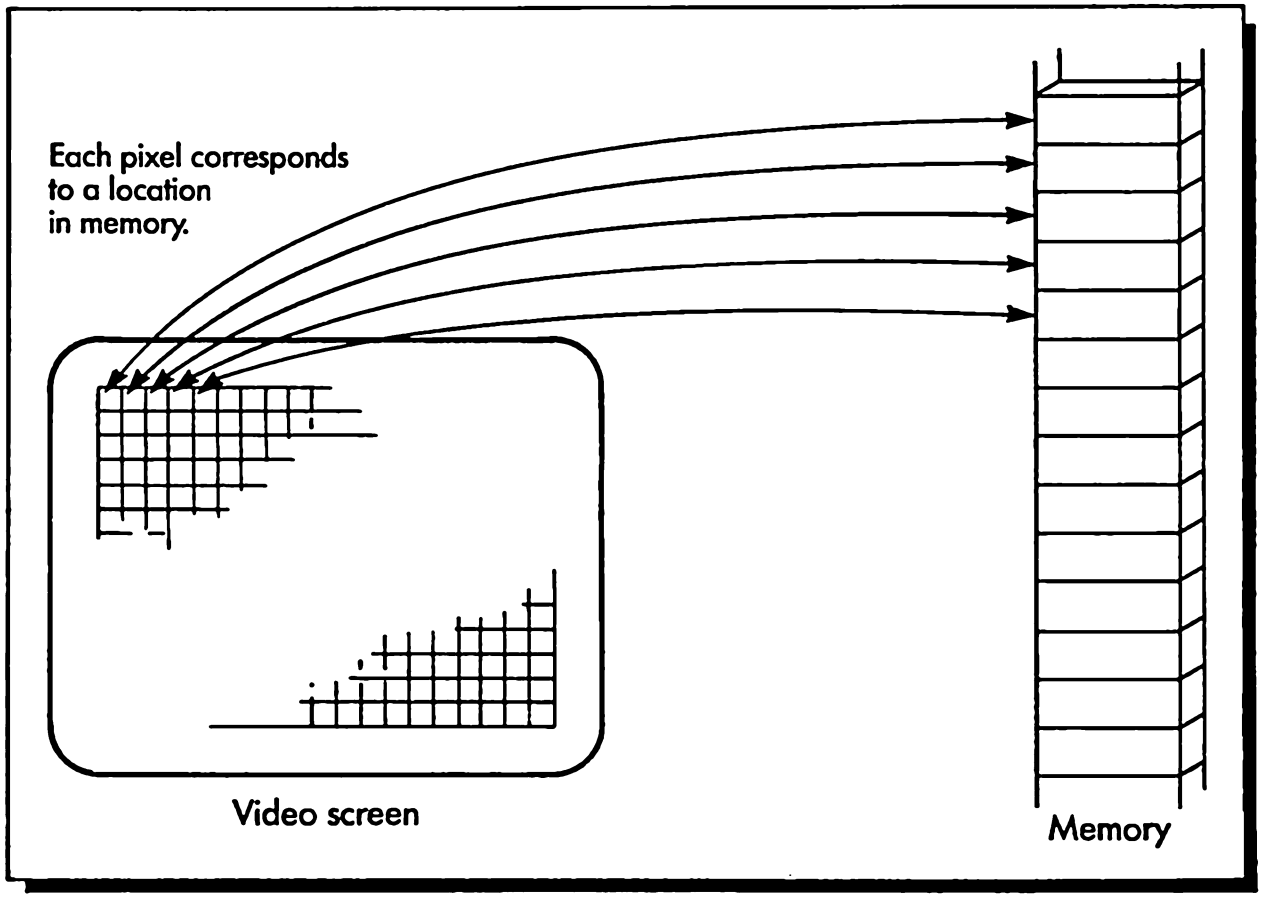


Figure 10-2. Memory-mapped graphics

screen. Then a space, and finally an "f " so only one line will be printed out:

```
-d bØØØ:Ø f
BØØØ:ØØØØ  2D Ø7 45 Ø7 75 Ø7 63 Ø7-6C Ø7 69 Ø7 64 Ø7 2Ø Ø7   -.E.u.c.l.i.d. .
            "-"   "E"   "u"   "c"   "l"   "i"   "d"
```

Well, would you look at that! There are those well-known letters, sitting in your computer's memory. Try the same procedure with another name: Plato, for instance. The name will appear in the same memory locations.

This should give you an idea what we mean by memory-mapped video. For every location on the screen, there is a corresponding location in memory. If you see a character on the screen, then you know you can look in memory and find the ASCII code that generated that character.

Attributes

Have you noticed something strange about the printout above? The ASCII characters that spell out "Euclid" occupy every *other* byte in memory, not every byte as you might expect. The odd-numbered bytes are all filled with 07h. Why is this?

In the last chapter we described the "attribute," a mysterious number which could cause a character on the screen to appear in reverse video, blinking, or intensified. As you recall, the attribute number 07h gave a "normal" character (green on black, non-blinking, etc.). Is it merely coincidence that the number 07h crops up in these two places?

The fact is that every character on the monochrome video screen is represented by *two* locations in memory. The *even* location holds the ASCII code for the character, and the *odd* location holds the attribute. Thus location B000:0000 contains the first ASCII character, and B000:0001 contains the attribute of the character, and so on. Now you know how it is that every character can have a different attribute.

Drawing on the Screen with DEBUG

If we can find out what's on the screen by examining memory with DEBUG, can we also *change* what's on the screen by inserting things in memory? Why not?

Get into DEBUG. We're going to make things a little easier on ourselves by changing the segment address in the DS register so that we don't have to keep typing in 8-digit addresses. The "D" and "E" commands in DEBUG always operate in reference to the DS register, so

by changing the address in this register we can change the segment these commands take as their "default" value. We set DS to B000h, which is the segment address of the monochrome video display in memory.

```
A>debug
-rds          ← DS register
DS Ø8F1       ← Old value
:bØØØØ        ← Enter segment address of monochrome display
```

Now hit the ↵ key enough times to scroll this off the top of the screen, so you're left with nothing but the DEBUG dash prompt "-" at the top of the screen. Then use the "E" command to insert some numbers into memory, as shown below:

```
-eØ           ← (Note: you don't need to type a segment address)
BØØØ:ØØØØ  2D.41   Ø7.fØ   3E.42   Ø7.87   2Ø.43   Ø7.f    2Ø.
              |       |       |       |       |       |       |
             "A"   Scan    "B"   Scan    "C"   Scan    Don't
                   code          code          code    type
                   for           for           for     ENTER
                   reversed,     blinking      intensified yet!
                   blinking
```

When you first type the "41" the letter "A" will appear in the upper left-hand corner of the screen. When you add the "f0" the "A" will immediately turn into reverse video and start flashing. The other letters and attributes will appear one by one as you type them in, as shown here:

```
ABC
|||└── Intensified
||└─── Blinking
|└──── Reverse video and blinking
```

How Big Is the Monochrome Memory?

The monochrome video memory contains 2000d character positions, each of which needs two bytes, for a total of 4000d bytes, or from 0 to F9Fh. Figure 10-3 shows how this section of memory is related to the characters on the screen.

You can check to see if this is true by using the "F" command to "fill" the memory with a constant and noting the effect on the screen. There is a small glitch with using "F" in this context. Since it fills *every* location, whatever we put into the even (ASCII) locations will also go into the odd (attribute) locations. But let's give it a try anyway. Type:

```
-fØ f9f 7
```

The screen will instantly be filled with little diamonds. This is the character the IBM PC puts on the screen when it sees a 07h. 07h is also the "normal" attribute, so the diamonds are green on black, non-blinking, and non-intensified. Try this:

```
-fØ f9f 7Ø
```

Now we get the screen filled with lowercase "p"s, in reverse video, since 70h is the ASCII for "p" and also the reverse video attribute.

The FILLS Program

Now you know how to fill the video memory with a constant using

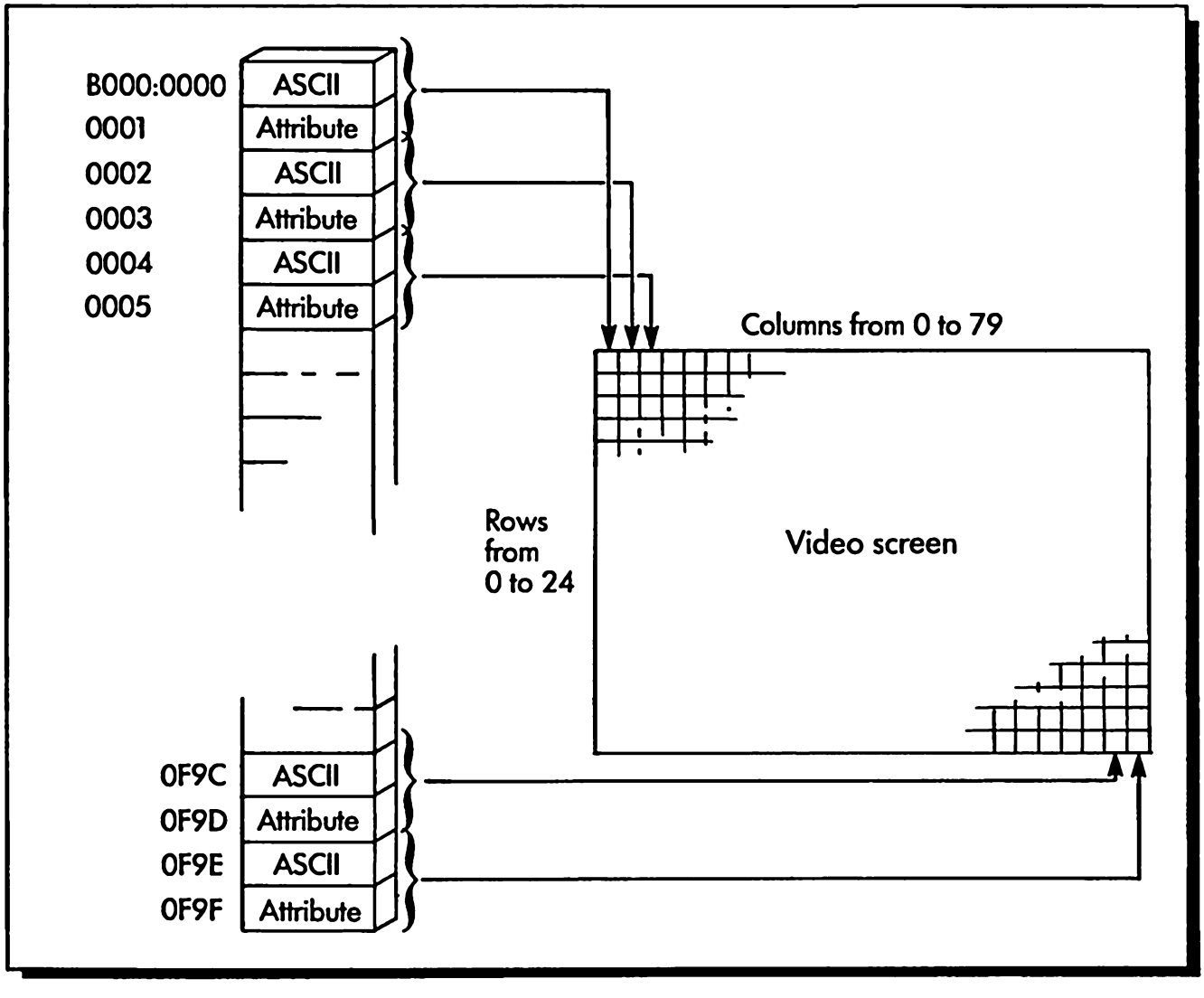


Figure 10-3. Monochrome memory and the video screen

DEBUG. What would an assembly-language program that did the same thing look like?

```
                                ;FILLS--Fills monochrome video memory
                                ;       with happy faces

                                ;*****************************************
                                ;define video memory as a segment

0000                            video   segment at 0b000h
0000  0800 [                            db      800h dup(?)
            ??
                ]

0800                            video   ends

                                ;*****************************************
0000                            coder   segment         ;define code segment

                                ;-----------------------------------------
0000                            fills   proc    far     ;main procedure

                                        assume cs:coder ;tell ASM where we are
                                        assume ds:video ;and where video is

                                ;set up stack for return
0000  1E                                push    ds      ;save old DS contents
0001  2B C0                             sub     ax,ax   ;make a zero
0003  50                                push    ax      ;save on stack

                                ;put the video memory segment in DS
0004  B8  ---- R                        mov     ax,video
0007  8E D8                             mov     ds,ax

                                ;set up to insert happy face, normal attribute
0009  B8 0701                           mov     ax,0701h ;happy face
000C  BB 0000                           mov     bx,0     ;start of buffer
000F  B9 0F9F                           mov     cx,0f9fh ;count

                                ;fill the screen with happy faces
0012  89 07                     again:  mov     [bx],ax ;insert char
0014  43                                inc     bx      ;incr pointer
0015  43                                inc     bx      ;  two bytes
0016  E2 FA                             loop    again   ;again

0018  CB                                ret             ;return to DOS
```

```
ØØ19                    fills   endp            ;end of procedure
                        ;-----------------------------------------
ØØ19                    coder   ends            ;end of code segment
                        ;*****************************************
                                end             ;end of assembly
```

Assemble and link FILLS into an EXE file. This program first changes the data segment to be the video memory at segment address B000h, just as we did above in DEBUG. Then it uses a loop, as we've seen in earlier programs, to fill memory up with happy faces.

Some things to notice about FILLS:

1. It uses a full word (not a byte) for the fill so it can insert both the ASCII character and the attribute at the same time. Then it increments the pointer (BX) twice.

2. It can't use an INT 20h to return from the program because the program itself alters the DS register. It must therefore save DS at the beginning of the program, and return with a RET.

Otherwise this program is fairly straightforward. How about something a little more ambitious?

The DRAW-1 Program

The program we're about to describe will permit you to draw pictures on the screen. It will also give you a good idea of the limitations of graphics in the monochrome mode. You'll see the kind of crude pictures that result when the lines are a whole character wide and high. On the other hand, the patterns you can create with this program have a certain modern-looking angularity to them. You might use this program to generate designs for company logos or high-tech wallpaper.

Type in the program, and assemble and link it into an EXE file. When you execute it, nothing will happen until you press one of the cursor control keys (which are on the numeric keypad). Then the "cursor" (actually the square graphics characters) will materialize and begin to move around the screen, leaving a trail of clones of itself wherever it goes. The result is a series of thick lines, with right angle turns. To exit from the program, type Ctrl C.

Type in the program, and assemble and link it into an EXE file.

```
                        ;DRAW-1--Program to draw on screen with
                        ;         cursor arrows. Uses ROM routines
= ØØØØ                  read_c  equ   Øh   ;read character code
= ØØ16                  key_rom equ  16h   ;ROM keyboard routine
```

```
= 0048                          up      equ   48h  ;scan code for up arrow
= 0050                          down    equ   50h  ;scan code for down arrow
= 004D                          right   equ   4dh  ;scan code for right arrow
= 004B                          left    equ   4bh  ;scan code for left arrow
= 00DB                          block   equ  0dbh  ;solid graphics character
= 0003                          ctrl_C  equ   3h   ;control-C (break) char
                                ;*********************************************
0000                            video   segment at 0b000h ;define extra seg
0000                            wd_buff label   word
0000  0FA0 [                    v_buff  db      25 * 80 * 2  dup (?)
            ??
                ]
0FA0                            video   ends
                                ;*********************************************
0000                            pro_nam segment         ;define code segment
                                ;---------------------------------------------
0000                            main    proc    far     ;main part of program
                                        assume  cs:pro_nam
                                        assume  es:video
0000                            start:          ;starting execution address
                                ;set up stack for return
0000  1E                                push    ds      ;save DS
0001  2B C0                             sub     ax,ax   ;set AX to zero
0003  50                                push    ax      ;put it on stack
                                ;set ES to extra segment
0004  B8  ---- R                        mov  ax,video
0007  8E C0                             mov  es,ax
                                ;clear screen by writing zeros to it
                                ;   even bytes get 0 (character)
                                ;   odd bytes get 7 (normal "attribute")
0009  B9 07D0                           mov  cx, 80 * 25        ;count
000C  BB 0000                           mov  bx,0               ;start of buff
000F  26: C7 87 0000 R 0700     clear:  mov  [wd_buff + bx], 0700h
0016  43                                inc  bx                 ;incr pointer
0017  43                                inc  bx                 ;  twice
0018  E2 F5                             loop clear              ;do again
                                ;screen pointer will be in CX register
                                ;   row    number (0 to 24d) in CH
                                ;   column number (0 to 79d) in CL
                                ;set screen pointer to center of screen
001A  B5 0C                             mov   ch,12d    ;# rows divided by 2
001C  B1 28                             mov   cl,40d    ;# columns div by 2
                                ;get character from keyboard, using ROM BIOS
                                ;  routine
001E                            get_char:
001E  B4 00                             mov   ah,read_c ;code for read char
0020  CD 16                             int   key_rom   ;keyboard I/O ROM call
0022  3C 03                             cmp   al,ctrl_C ;is it control-C?
0024  74 2F                             jz    exit      ;yes
```

```
0026  8A C4                          mov   al,ah     ;put scan code in AL
0028  3C 48                          cmp   al,up     ;is it UP arrow?
002A  75 02                          jnz   not_up    ;no
002C  FE CD                          dec   ch        ;yes, decrement row
002E                           not_up:
002E  3C 50                          cmp   al.down   ;is it DOWN arrow?
0030  75 02                          jnz   not_down  ;no
0032  FE C5                          inc   ch        ;yes, increment row
0034                           not_down:
0034  3C 4D                          cmp   al.right  ;is it RIGHT arrow?
0036  75 02                          jnz   not_right ;no
0038  FE C1                          inc   cl        ;yes. increment column
003A                           not_right:
003A  3C 4B                          cmp   al,left   ;is it LEFT arrow?
003C  75 02                          jnz   lite_it   ;no
003E  FE C9                          dec   cl        ;yes, decrement column
0040                           lite_it:
0040  B0 A0                          mov   al,160d   ;bytes per row into AL
0042  F6 E5                          mul   ch        ;times # of rows
                                                     ;  result in AX
0044  8A D9                          mov   bl,cl     ;# of columns in BL
0046  D0 C3                          rol   bl,1      ;times 2 to get bytes
0048  B7 00                          mov   bh,0      ;clear top part of BX
004A  03 D8                          add   bx,ax     ;add AX to it
                                                     ; gives address offset
                               :address offset in BX.  Put block char there
004C  B0 DB                          mov   al,block
004E  26: 88 87 0000 R               mov   [v_buff + bx],al
0053  EB C9                          jmp   get_char  ;go get next arrow
0055  CB                       exit:  ret             ;return to DOS
0056                           main   endp    ;end of main part of program
                               ;-----------------------------------------
0056                           pro_nam ends    ;end of code segment
                               ;*****************************************
                                     end    start   ;end assembly
```

This program uses the keyboard ROM call we described in the last section to figure out which of the cursor control keys are being pressed. Notice that it doesn't matter how the (Num Lock) key is toggled; the scan codes are the same whether these keys are in "Edit Mode" or "Numeric Mode," so our program doesn't need to worry about that.

For a change, the program uses the Extra Segment instead of the Data Segment to reference the video memory. Why? Well, sometime you might want to modify this program, and use some data in it (maybe to print a message). To do this, it would be more convenient to put the data in the data segment and save the video memory for the extra segment, so that's what we've done. The program loads the segment address "video"

into the ES register, and uses an ASSUME statement to tell the assembler that's what it did.

In the last chapter we showed you how to clear the screen using the Scroll Up ROM call. Here we clear it in a more direct way, by writing zeros into all the ASCII video memory locations. We also put 07h into all the attribute positions. This is the most direct and the fastest way to clear the screen.

The LABEL Pseudo-op

In the Extra Segment we show the following section of code:

```
0000                    wd_buff         label   word
0000  0FA0 [            v_buff db       25 * 80 * 2  dup (?)
            ??
               ]
```

What does this mean? Well, first we want to define a buffer of 25d times 80d times 2 bytes, which is the number of bytes in the monochrome memory. We need it defined in terms of *bytes* because, when we move our graphics character around the screen later in the program, we aren't going to change the attribute bytes at all; we'll simply change the ASCII (even) byte for the appropriate location.

On the other hand, when we clear the screen, it's more efficient to think of the screen in terms of *words*. That way we can write 80 * 25 words into the memory, instead of 80 * 25 * 2 bytes, which would take twice as long.

LABEL Pseudo-op

Defines (or redefines) the type of a variable or location name.

```
WORD_VAR  DW  1234h       ;defines location as word
BYTE_VAR  LABEL  BYTE     ;defines first byte of
                          location as byte
WORD_VAR  LABEL  WORD     ;defines location as word
BYTE_VAR  LABEL  BYTE     ;defines location as byte
```

This pseudo-op ~~is typically~~ used when a location needs to be defined as two different types at the same time.

Data types: BYTE, WORD, DWORD

Code types: NEAR, FAR

This is where the LABEL pseudo-op comes in. It lets us define the same location in memory in two different ways. First, using the DB pseudo-op, we define the locations in the buffer WD_BUFF as bytes. Our assembly language instructions can now reference this block of memory as bytes, but not as words. Then, using the LABEL pseudo-op, we *also* define the locations as words. Now our program can refer to a location in the buffer either as a byte, or as a word.

The DB (or DW) and LABEL pseudo-ops are similar in that they both define memory locations to be of a certain type, but DB and DW actually *set aside the specified bytes or words in memory,* while LABEL only *names* and sets the type of a particular memory location.

> The assembler knows the difference between words and bytes, and makes sure you know it too.

Type Checking

The MACRO Assembler needs the LABEL pseudo-op in its bag of tricks because it is so "hard-line" about data types. For instance, if you've defined a variable to be a word with a DW pseudo-op, then you are in big trouble with the assembler if you ever try to treat the variable as a byte or as a double-word. For instance, if your program has the statement

```
word_var  dw   1234h
```

and later in your program you want to get the 8-bit part of this, 34h, into the AL register, you can't just say

```
mov  al,word_var
```

because the assembler will recognize that you are trying to put a 16-bit *word* into a byte-sized register. In order to save you from what it assumes is your own stupidity, it flags this as an error. If you really want to refer to the same location in two different ways, then you must use LABEL to *define* it in two different ways.

Operating DRAW-1

At the heart of the program are two numbers which define the current location of the "cursor" (the solid block character) on the screen. The row number (from 0 to 24d) is kept in the CH register, and the column number (from 0 to 79d) is kept in the CL register. At the start of

the program these two values are set to put the cursor in the center of the screen.

This section of the program has two parts. The first part, from "get_char" down to "not_right", is concerned with reading the keyboard to see which of the four cursor movement keys the user has pushed, and then changing the values in CH and CL to reflect these values. Thus, if the program finds that the left arrow has been pushed, it decrements the column number in CL. The ROM keyboard routine, called with INT 16h, is used to read the keyboard, since we want to get the scan codes for the cursor-motion keys. (This ROM routine was described in the last chapter.)

The second part of the program then "paints" the block character at whatever location CH and CL specify. The program continuously cycles through this loop, checking the keyboard and displaying the block character at the current cursor location. If the cursor has not moved since the last display, it is displayed again at the same place.

The trick here is to translate the row and column numbers in CH and CL into a single offset address in BX. Once we have the offset address in BX, it's easy to use indirect addressing to write the block graphics characters to this location in memory. To make an address out of the numbers in CH and CL, we first multiply the number of bytes per row (160d) by the row number in CH. Then we multiply the column number in CL by 2, since there are two bytes for each location. Finally we add these two results to obtain the offset address. This is all done in the six instructions following the "lite_it" label. The block graphics character is then written into memory at this address with the indirect addressing instruction

```
mov  [v_buff + bx],al
```

You can probably think of a lot of changes and improvements you could make to this program. You might want to try drawing with a graphics character other than the block; an asterisk, for example. Or you could add an "erase" feature to the program by writing blanks to the screen if a certain keyboard key was pressed while moving the cursor.

Dissolve: Monochrome to Color

In the next section we're going to move on to color graphics. If you don't have the "color card" installed in your set you won't be able to try out the programs in the rest of this chapter. However, you might want to read the section anyway. You'll find out how the graphics display differs

from monochrome, and how to draw dots and lines. In addition you'll learn a new 8088 instruction, a new pseudo-op, and more about how segments operate.

Color Graphics

Now we'll introduce you to the exciting world of color graphics. Actually, this topic is so varied and can lead in so many directions that we can only scratch the surface. Whole books can be written about graphics on the IBM PC. (An excellent one is *Graphics Primer for the IBM PC*, by Mitchell Waite and Christopher L. Morgan [Berkeley: Osborne/McGraw-Hill, 1983].) What we hope to do here is show you the fundamentals, so you can get started writing your own graphics programs. We'll leave the more far-out applications for other books.

What's the Difference Between Monochrome and Color?

How does the color display differ from monochrome (besides being in color)? First of all, the color display has a much higher resolution, and thus must use much more memory. As we mentioned, the color adapter board provides two choices: 320x200 medium resolution, with four different colors; and 640x200 high resolution, in black and white. In this discussion we're going to concentrate on the 320x200 color display. The 640x200 is actually very similar — by understanding one you will understand the other.

Changing Modes

Let's assume that you use the monochrome display for programming, but that you also have your color board installed and your monitor or TV set hooked up and ready to go. (If you are using only the color display, then you will not need the information in this section.) The question is: How do you get the computer to send its output to the color display instead of to the monochrome display?

If you have DOS 2.00 it's easy to change graphics modes: there is a DOS command called MODE which will do it for you. For instance, to get into the medium resolution color mode, you would enter

```
A>mode co40
```

There are many other options with the MODE command. You can read about them in the *IBM Personal Computer Disk Operating System* manual.

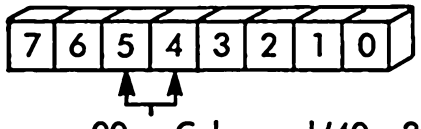
However, to use MODE you have to be a human being, typing from the keyboard. Your program can't access this feature. Also, if you are using DOS 1.10, MODE will not have the capability to change the display. It is therefore useful to know how to write a program which will change graphics modes. Also, understanding how such a program works will increase our understanding of how the IBM PC handles graphics.

To change graphics modes, our program needs to do two things:

1. Alter some bits in a special byte in memory called the "equipment flag."
2. Tell the ROM BIOS video routine what display mode it wants.

The Equipment Flag

If you look through the ROM BIOS listing in your *IBM Personal Computer Technical Reference* manual, you'll find a section called "ROM BIOS Data Areas." The segment address of this section of code is 40h. At offset address 10h in this segment is the equipment flag. (That's absolute address 00410h.) When the PC is first turned on, a ROM BIOS routine reads the settings of your peripheral hardware switches and stores their on/off states in the equipment flag location. If we want to change displays, we must "fool" the BIOS routines into thinking that the switch settings have changed. We do this by putting new values in certain bits of this memory location.



00 = Color card (40 x 25 characters)
10 = Color card (80 x 25 characters)
11 = Monochrome display

Specifically, we change bit 4 from 1 to 0 if we want to change from monochrome to color. Bit number 5 is always set on, unless we want to change to the 40 character mode, which means that text will appear in forty columns on the screen instead of 80. The other bits do not apply to the video displays at all, but their settings must not be lost when bits 4 and 5 are changed.

ROM BIOS Video Routine

Now that you have chosen the active display using the equipment flag, you need to control the graphics mode. If the color card is selected you do this with a call to a ROM BIOS routine.

Go back to the ROM BIOS listing and you'll find a "Set Mode" function under the Video_IO section (INT 10h). There are a number of different options here, but we're only going to talk about three of them:

1. 80x25 black and green text (the regular monochrome display). Called with AL=2.
2. 640x200 black and white (high resolution). Called with AL=6.
3. 320x200 color (medium resolution). Called with AL=4.

To activate one of these modes your program must do an INT 10h, with AH=0 (to select the "Set Mode" function) and AL set as shown above. This will tell the ROM BIOS what mode you want to be in, and all subsequent output to the screen will be directed accordingly.

The CHAMODE Program

The CHAMODE program gives you three choices when you first execute it:

Type "m" for "monochrome," normal black and green display.

Type "h" for "high resolution," 640x200 black and white.

Type "c" for "color," 320x200.

When you make your selection it then modifies the Equipment Flag and sends the appropriate message to ROM BIOS, thus switching your system into the desired mode. Type in the program, assemble, and link it. Also, convert it into a COM file with EXE2BIN. For variety and fast loading we've used the COM file format in this program.

```
                        ;CHAMODE--Program to change screen modes

= 0001                  key_in  equ     1h      ;keyboard input
= 0009                  pstring equ     9h      ;print string
= 0021                  doscall equ     21h     ;DOS interrupt number

                        ;*****************************************

                        ;SEGMENT TO CONTAIN EQUIPMENT FLAG

0000                    rom_da  segment at 40h
0010                            org     10h
0010  ????              eq_flag dw      ?
0012                    rom_da  ends
                        ;*****************************************

0000                    codeseg segment         ;define code segment
```

```
                                        assume cs:codeseg
                                        assume ds:codeseg

0100                                    org 100h
0100                            start:

                                ;print intro message
0100  BA 015F R                         mov  dx,offset mess  ;addr of mess
0103  B4 09                             mov  ah,pstring ;print string function
0105  CD 21                             int  doscall    ;call DOS

                                ;set data segment to equipment flag
                                        assume ds:rom_da
0107  B8  ---- R                        mov  ax,rom_da  ;set DS to
010A  8E D8                             mov  ds,ax      ; equipment flag

                                ;get input letter
010C  B4 01                             mov  ah,key_in  ;keyboard input
010E  CD 21                             int  doscall    ;call DOS
0110  3C 6D                             cmp  al,'m'     ;is it "m" ?
0112  74 0A                             je   mono       ;monochrome
0114  3C 68                             cmp  al,'h'     ;is it "h" ?
0116  74 1B                             je   hi_res     ;hi res blk and white
0118  3C 63                             cmp  al,'c'     ;is it "c" ?
011A  74 2C                             je   color      ;320x200 color
011C  EB E2                             jmp  start      ;unknown input

                                ;SET UP FOR MONOCHROME DISPLAY
011E                            mono:
011E  A1 0010 R                         mov  ax,eq_flag   ;get equipment flag
0121  25 00CF                           and  ax,1100111lb ;mask off video bits
0124  0D 0030                           or   ax,0011000b;monochrome bits
0127  A3 0010 R                         mov  eq_flag,ax   ;back into flag

012A  B0 02                             mov  al,2       ;80 column b & w code
012C  B4 00                             mov  ah,0       ;"setmode" function
012E  CD 10                             int  10h        ;call Video BIOS
0130  EB 2B 90                          jmp  exit

                                ;SET UP FOR 640 x 200 BLACK AND WHITE
0133                            hi_res:
0133  A1 0010 R                         mov  ax,eq_flag   ;get equipment flag

0136  25 00CF                           and  ax,1100111lb ;mask off video bits
0139  0D 0020                           or   ax,0010000b ;color card 80 x 25
013C  A3 0010 R                         mov  eq_flag,ax   ;back into flag
```

```
013F  B0 06                       mov  al,6       ;640x200 bl & white
0141  B4 00                       mov  ah,0       ;"set mode" function
0143  CD 10                       int  10h        ;call Video BIOS
0145  EB 16 90                    jmp  exit

                                ;SET UP FOR 320 x 200 COLOR MODE
0148                            color:
0148  A1 0010 R                   mov  ax,eq_flag   ;get equipment flag
014B  25 00CF                     and  ax,1100111lb ;mask off video bits
014E  0D 0020                     or   ax,0010000b ;color card 80 x 25
0151  A3 0010 R                   mov  eq_flag,ax   ;back into flag

0154  B0 04                       mov  al,4       ;320x200 color
0156  B4 00                       mov  ah,0       ;"set mode" function
0158  CD 10                       int  10h        ;call Video BIOS
015A  EB 01 90                    jmp  exit

015D                            exit:
015D  CD 20                       int  20h        ;return to DOS

                                ;SIGN-ON MESSAGE

015F  0D 0A                     mess db  13,10
 0161  54 79 70 65 20 22              db  'Type "m" for 80x25 monochrome',13,10
       6D 22 20 66 6F 72
       20 38 30 78 32 35
       20 6D 6F 6E 6F 63
       68 72 6F 6D 65 0D
       0A
0180  20 20 20 20 20 22              db  '     "h" for 640x200 b & w',13,10
       68 22 20 66 6F 72
       20 36 34 30 78 32
       30 30 20 62 20 26
       20 77 0D 0A
019C  20 20 20 20 20 22              db  '     "c" for 320x200 color',13,10
       63 22 20 66 6F 72
       20 33 32 30 78 32
       30 30 20 63 6F 6C
       6F 72 0D 0A
01B8  53 65 6C 65 63 74              db  'Selection: $'
       69 6F 6E 3A 20 24
01C4                            codeseg ends
                                ;*********************************************

                                   end  start
```

The SEGMENT AT Expression Pseudo-op

There are a few new things to notice about the CHAMODE program. First, it uses a new form of the SEGMENT pseudo-op:

```
segment at 40h
```

In this case we know the absolute address where we want the segment to begin, so we can specify it explicitly. Now when DOS (or DEBUG) loads the program it has no choice about where to put this segment: the address has already been specified. The equipment flag has an address of 10h in this segment, so the program sets that up with an ORG pseudo-op in the usual way. We can find the segment and offset addresses to use for the equipment flag by looking in the ROM BIOS listing and using the same addresses used by the program there.

Another thing to notice is that since this program is a COM file, we've put the sign-on messages in the same segment as the code. We had to do this, because there can only be one segment in a COM file. But, you may ask, what about the segment with the equipment flag? That's a separate segment — how can it work in a COM file? The answer is that it's all right to use more than one segment in a COM file, provided it has been given an absolute address with the SEGMENT AT pseudo-op.

> You can have more than one segment in a COM file, provided the additional segments are defined with SEGMENT AT.

Operating CHAMODE

The program is fairly straightforward. After printing the sign-on message, it reads the keyboard with a DOS function to find which of the three modes has been selected, then it branches to one of three very similar routines. In each routine the appropriate bit or bits are ORed onto the Equipment Flag, without disturbing the other bits. Then the Video ROM routine is called, with AL set to the appropriate number, as described above.

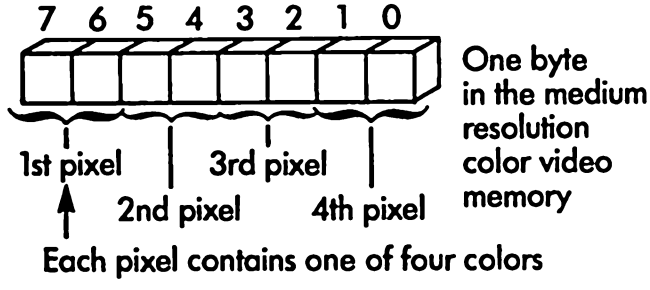
Before referencing the Equipment Flag, the DS register has to be loaded with the segment address where this location is, namely 40h. But we can't do that until after we print the messages, since the messages are in the code segment, and DS must contain the address of the code segment to reference the messages. Once the messages are printed, we can change DS.

Exploring Color Graphics with DEBUG

Now that you can switch back and forth between different display modes using the CHAMODE program, let's use DEBUG to investigate some of the characteristics of the color graphics display. In this section we'll be talking about the 320x200 color mode, not the 640x200 black and white mode.

The Color Graphics Memory Map

As you recall, in the monochrome display, two bytes of memory were allotted to each character position on the screen: one for the ASCII code, and one for the attribute. In the color mode the correspondence between memory and the video screen is somewhat different. Instead of two *bytes* being assigned to each position on the screen, only two *bits* are. This is because there are so many more screen locations to assign. In the monochrome mode there were only 25 rows times 80 columns, for a total of 2000 screen locations. In the color mode there are 320 times 200 screen locations, or pixels. But 320 times 200 is 64,000d pixels! Assigning even one byte to each of these locations would use up too much memory, so each byte is used to represent *four* pixels, as shown here:



Two bits represent four things, so there is a choice of four different colors for each pixel. One of these colors is usually black, so that we can have blank, uncolored areas on the screen. This leaves three numbers for colors, as shown:

00 = black

01 = yellow

10 = blue

11 = white

Experimenting with DEBUG in the Color Memory

Let's get into DEBUG and see if we can manipulate the color

memory. If you're using the monochrome display, the first thing you want to do is call up CHAMODE, and use it to switch to the color display (or you can use the MODE command from DOS).

```
A>chamode

type "m" for 80x25 monochrome
     "h" for 640x200 b & w
     "c" for 320x200 color
Selection: c                    <— Type this to get into color
```

Then, on the color screen, call up DEBUG.

It's more convenient, as we learned when we investigated the monochrome memory, to set the data segment register to the segment address of the color memory, so we'll start off doing that:

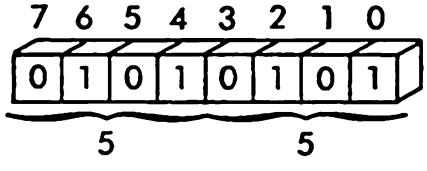
```
A>debug
-rds
DS 08F1
:b800                   <— Set the data segment to the color memory
```

Now, try typing the following:

```
-f0 400 55              <— Turn first 400h * 4 locations yellow
```

A dozen yellow lines appear at the top of the screen! At least, they may be yellow — it depends how your TV is adjusted, and even what kind of TV set or monitor you have.

What we've done is to use "F" to fill in the first 400h locations of the color memory with a constant, 55h. Let's see what this looks like on the bit level:



As you can see, 55h is 01010101 binary. If we insert commas, the binary number should be easier to read: 01,01,01,01. This will put a 1 in each of the four color locations, thus giving them all the same color, number 1, which should be yellow.

There are two more colors to go. Let's make the next dozen lines blue:

```
-f 4ØØ 8ØØ aa          <— Turn next 400h * locations blue
```

AAh is 10,10,10,10 binary, which puts a 10 in each pixel, the code for blue.

Finally, the third color can be used, which will give us a dozen lines of white:

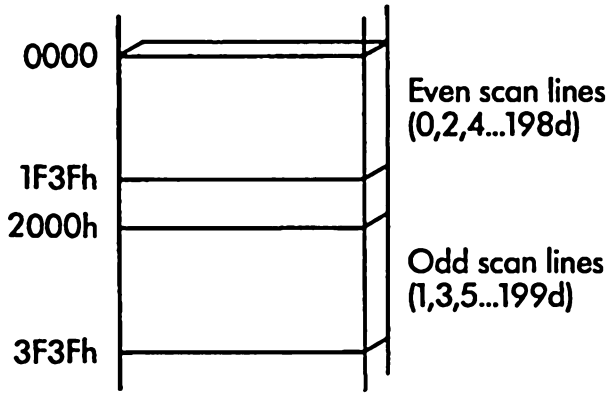
```
-f 8ØØ cØØ ff          <— Turn next 400h * 4 locations white
```

Here we get an 11 — the code for white — in each pixel location.

The Two Halves of the Color Memory

You may have noticed something else about the lines that "F" placed on the screen: only half of them are there. That is, only the *even* scan lines are filled in. The odd ones, in between, have not been turned on. Why is this?

It turns out that the color memory in the IBM PC is actually divided into two parts, one for the even scan lines and one for the odd. This arrangement was adopted to make it easier for the video hardware to write the picture to the screen. It looks like this:



So if you want to see the lines between the lines filled in, you need to start at the top of the second part of the color memory and repeat the three steps above. Try it:

```
-f 2ØØØ 24ØØ 55        <— Fill in the odd lines with yellow
-f 24ØØ 28ØØ aa        <— Fill in the odd lines with blue
-f 28ØØ 2cØØ ff        <— Fill in the odd lines with white
```

As you can see, filling in an area with a solid block of color requires

two passes, one for the even lines, and one for the odd. Figure 10-4 shows the relationship of the color screen to video memory.

Tired of all those colors? You can erase the entire screen by filling it with zeros:

```
-fØ 3f3f Ø          ← Clear entire screen
```

Another easy thing to do is to create dotted lines. If, say, two of the pixels in a byte are set to color values, and the next two are set to zero, the result will be a dotted line. The hex number A0 is 10,10,00,00 in

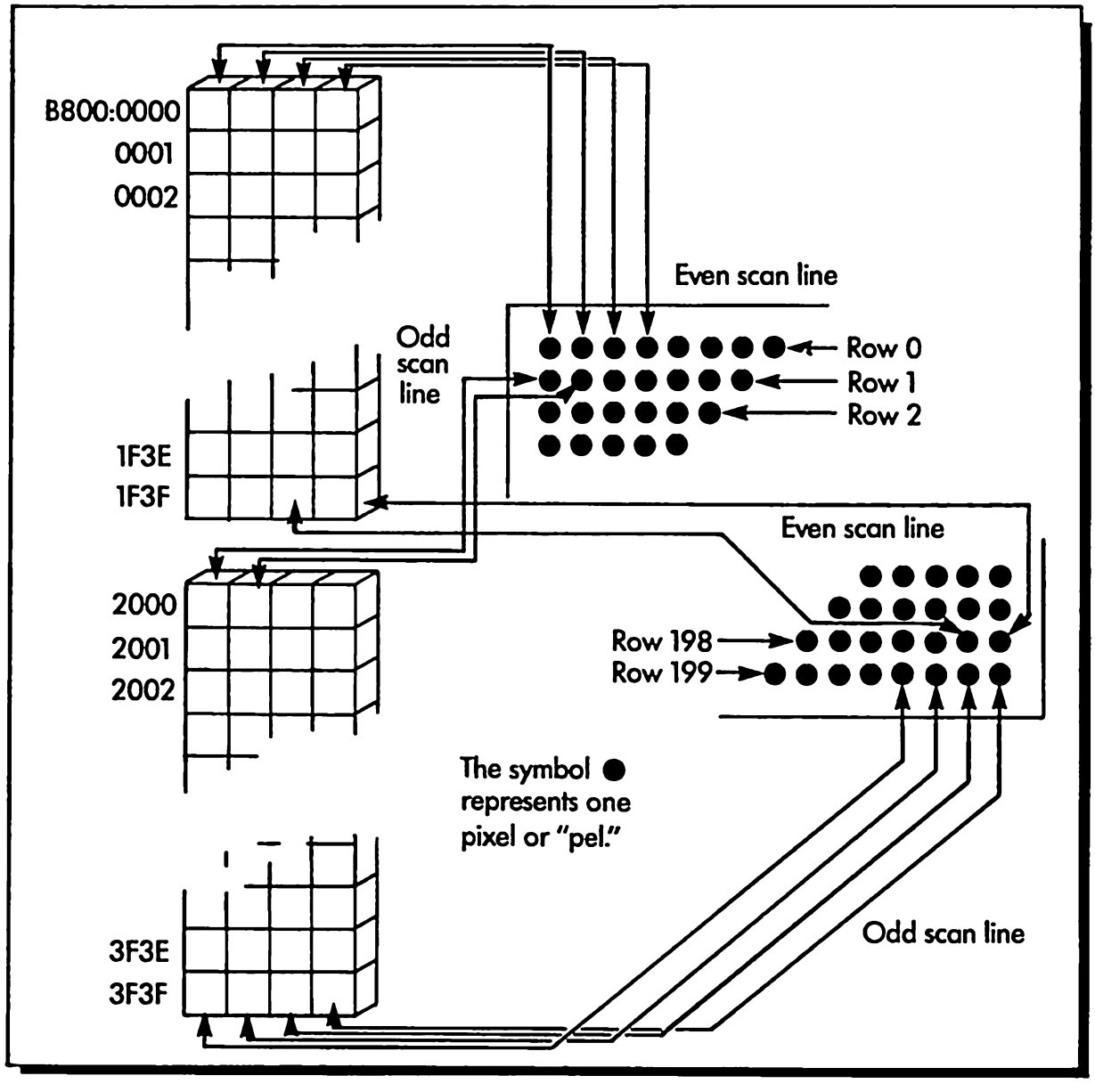


Figure 10-4. Color memory and the video screen

binary, which will give us two blue dots followed by two spaces:

```
-fØ 4ØØ aØ              ←— Dotted blue line
```

BASIC 2.00 offers an option called STYLE, in the LINE statement, which will achieve some of these effects, but again assembly language gives you a far greater range of possibilities.

It's fun to play around with various patterns, such as alternating solid and dotted lines for a textured effect, but we'll tear ourselves away from that sort of thing at this point, and write some programs.

DRAW-CO: The DRAW Program in the Color Mode

The next program is a variation of the DRAW program we wrote earlier for the monochrome screen. As with the monochrome version, you can cause a line to be drawn in any of the four directions on the screen by pressing the cursor keys. However, this program, DRAW-CO (for "COlor") has an added feature. Before you start to draw with the cursor keys, you must press a number key from 0 to 3. Pressing 1, 2, or 3 will cause the line to be colored in one of the three colors. Pressing 0 will cause the line to disappear — that is, the location of the moving dot will change, but you won't be able to see it, since it will be drawing a black line on a black background. To exit from the program and return to DOS, press the Esc key.

Type in the program, assemble it into an EXE file, and try it out. (Be sure to toggle the Num Lock key correctly.) Nothing will appear to happen until you press an arrow key; then the moving line will appear. You'll learn some interesting things about color graphics. One of the first things you may discover if you have a TV set is that not all vertical lines produce the same color! This is a result of the way the TV set is made: because of limitations in the TV circuitry, vertical lines, even drawn in the same color, will have different colors in different locations. This problem can make the creation of sophisticated pictures on the TV screen something of a challenge. A color monitor does not have this problem.

Here's the DRAW-CO program:

```
                        ;DRAW-CO--Program to draw on screen with
                        ;         cursor arrows. Uses ROM routines

                        ;For 32Ø x 2ØØ medium res color mode
= ØØ48                  up      equ   48h  ;scan code for up arrow
= ØØ5Ø                  down    equ   5Øh  ;scan code for down arrow
= ØØ4D                  right   equ   4dh  ;scan code for right arrow
```

```
= ØØ4B                          left    equ   4bh  ;scan code for left arrow
= ØØ1B                          escape  equ   1bh  ;"escape" character
                                ;*****************************************

ØØØØ                            pro_nam segment         ;define code segment

                                ;-----------------------------------------
ØØØØ                            main    proc    far     ;main part of program

                                        assume  cs:pro_nam

ØØØØ                            start:          ;starting execution address

                                ;set up stack for return
ØØØØ  1E                               push    ds      ;save DS
ØØØ1  2B CØ                            sub     ax,ax   ;set AX to zero
ØØØ3  5Ø                               push    ax      ;put it on stack

                                ;clear screen by scrolling it, using ROM call

ØØØ4  B4 Ø6                            mov  ah,6       ;scroll up function
ØØØ6  BØ ØØ                            mov  al,Ø       ;code to blank screen
ØØØ8  B9 ØØØØ                          mov  cx,Ø       ;upper left = Ø,Ø
ØØØB  B2 4F                            mov  dl,79      ;lower right corner
ØØØD  B6 18                            mov  dh,24      ;  at 79,24
ØØØF  CD 1Ø                            int  1Øh        ;call video interrupt

                                ;screen pointer will be in CX, DX registers
                                ;   row    number (Ø to 2ØØd) in DX
                                ;   column number (Ø to 32Ød) in CX

                                ;set screen pointer to center of screen
ØØ11  BA ØØ64                          mov   dx,1ØØd   ;# rows divided by 2
ØØ14  B9 ØØAØ                          mov   cx,16Ød   ;# columns div by 2
                                ;get character from keyboard, using ROM BIOS
                                ;  routine

ØØ17                            get_char:
ØØ17  B4 ØØ                            mov   ah,Ø      ;code for read char
ØØ19  CD 16                            int   16h       ;keyboard I/O ROM call
ØØ1B  3C 1B                            cmp   al,escape ;is it escape char?
ØØ1D  74 2A                            jz    exit      ;yes

ØØ1F  3C 33                            cmp   al,33h    ;is it more than "3"
ØØ21  7F Ø8                            jg    plot_it   ;yes, not a color
ØØ23  3C 3Ø                            cmp   al,3Øh    ;is it less than "Ø"
ØØ25  7C Ø4                            jl    plot_it   ;yes, not a color
```

```
0027  8A D8                   mov   bl,al     ;save it in BL
0029  EB EC                   jmp   get_char  ;get next character

                        ;figure out which way to go, and draw new line
002B                    plot_it:
002B  8A C4                   mov   al,ah     ;put scan code in AL
002D  3C 48                   cmp   al,up     ;is it UP arrow?
002F  75 01                   jnz   not_up    ;no
0031  4A                      dec   dx        ;yes, decrement row
0032                    not_up:
0032  3C 50                   cmp   al,down   ;is it DOWN arrow?
0034  75 01                   jnz   not_down  ;no
0036  42                      inc   dx        ;yes, increment row
0037                    not_down:
0037  3C 4D                   cmp   al,right  ;is it RIGHT arrow?
0039  75 01                   jnz   not_right ;no
003B  41                      inc   cx        ;yes, increment column
003C                    not_right:
003C  3C 4B                   cmp   al,left   ;is it LEFT arrow?
003E  75 01                   jnz   lite_it   ;no
0040  49                      dec   cx        ;yes, decrement column

                        ;use ROM routine to write dot
                        ;requires row # in DX, col in CX, color in AL
0041                    lite_it:
0041  8A C3                   mov   al,bl     ;set color value
0043  B4 0C                   mov   ah,12d    ;write dot function
0045  CD 10                   int   10h       ;video BIOS routine

0047  EB CE                   jmp   get_char  ;go get next arrow

0049  CB                exit:  ret             ;return to DOS

004A                    main   endp    ;end of main part of program
                        ;-----------------------------------------
004A                    pro_nam ends   ;end of code segment
                        ;*****************************************

                               end     start   ;end assembly
```

The Write Dot ROM BIOS Routine

The program is fairly easy to understand: it is in many ways similar to the earlier monochrome DRAW program. The most important difference is that it makes use of a ROM BIOS routine called Write Dot. This is one of the functions of the video I/O section of ROM. Write Dot is entered with the following registers set:

AH = 12h

DX = row number of dot to be displayed

CX = column number of dot to be displayed

AL = color value of the dot

Then the usual INT 10h is executed. Of course, before this routine is used, the display has to be switched to the graphics mode (with CHAMODE or MODE).

The use of this ROM routine to plot points is a great convenience, since it saves having to write the code necessary to figure out where the dot is going to go (a process which is complicated by having to deal with the two banks of memory for odd and even scan lines, as we'll soon see).

The program always keeps the current values of the pixel's row and column available in the DX and CX registers respectively. When a cursor control key is pressed, the value in DX or in CX will change. When [↑] is pressed, the value of DX is decremented; when [↓] is, the value is incremented. When [←] is pressed, the value of CX is decremented, when [→] is, the value is incremented.

Since AL is used for arithmetic in the course of the program, BL is used to hold the current color value. This value is moved into AL just before the ROM routine is called.

DRAW2CO: Using a Subroutine to Plot the Dot

It's interesting to imagine just what the ROM BIOS Write Dot routine does. Essentially it has to translate the row and column numbers in DX and CX into a memory address in the video memory. The segment address of this part of memory is B800, and — as we learned — there are two separate "banks" of memory: one from offset address 0 to 1F3Fh for the even scan lines, and one from 2000h to 3F3F for the odd scan lines. The fact that there are two banks complicates the calculation of the address.

Also, since there are four pixels in every byte, the program must figure out not only the address to be modified, but which of the four two-bit color values within this byte is to be modified.

Let's take the DRAW-CO program and modify it to use its own subroutine to plot the dot. This will provide experience in the common programming problem of accessing the graphics memory, and at the same time give you an appreciation of just how much hard work the ROM BIOS routines do for you when you call them. We'll call the program DRAW2CO.

What modifications do we need to make to DRAW-CO so that it can

use its own subroutine (besides writing the routine itself)? Since the program must access the video memory directly, it must declare a segment to correspond to it. We'll use the Extra Segment (so that the program may use a separate data segment in possible future modifications). We must tell the assembler about this segment with the ASSUME pseudo-op, and (don't forget!) actually put the segment address of the video segment into the ES register. This is done in addresses 0004 and 0007 of DRAW2CO. Then we must remove the instructions that called the ROM BIOS Write Dot routine, and substitute a CALL to our own subroutine, which we'll call PLOTSUB. (Again, remember to toggle (Num Lock) correctly.)

Here's the program listing for DRAW2CO:

```
                        ;DRAW2CO--Program to draw on screen with
                        ;         cursor arrows. Uses internal
                        ;         subroutine to plot dot

                        ;For 32Ø x 2ØØ medium res color mode
                        ;
= ØØ48                  up      equ   48h  ;scan code for up arrow
= ØØ5Ø                  down    equ   5Øh  ;scan code for down arrow
= ØØ4D                  right   equ   4dh  ;scan code for right arrow
= ØØ4B                  left    equ   4bh  ;scan code for left arrow
                        ;
= ØØ1B                  escape  equ   1bh  ;"escape" character
                        ;
                        ;*********************************************

ØØØØ                    video   segment at Øb8ØØh ;define extra seg
ØØØØ                    wd_buff label   word
ØØØØ                    v_buff  label   byte
ØØØØ                    video   ends
                        ;*********************************************

ØØØØ                    pro_nam segment         ;define code segment

                        ;---------------------------------------------
ØØØØ                    main    proc    far     ;main part of program

                                assume  cs:pro_nam,es:video

ØØØØ                    start:          ;starting execution address

                        ;set up stack for return
ØØØØ  1E                       push    ds      ;save DS
ØØØ1  2B CØ                    sub     ax,ax   ;set AX to zero
ØØØ3  5Ø                       push    ax      ;put it on stack
```

```
                                ;set extra segment to video memory
0004  B8  ---- R                        mov  ax,video   ;get video address
0007  8E C0                             mov  es,ax      ;  put in ES

                                ;clear screen by scrolling it, using ROM call

0009  B4 06                             mov  ah,6       ;scroll up function
000B  B0 00                             mov  al,0       ;code to blank screen
000D  B9 0000                           mov  cx,0       ;upper left = 0,0
0010  B2 4F                             mov  dl,79      ;lower right corner
0012  B6 18                             mov  dh,24      ;  at 79,24
0014  CD 10                             int  10h        ;call video interrupt

                                ;screen pointer will be in CX, DX registers
                                :   row    number (0 to 200d) in DX
                                :   column number (0 to 320d) in CX

                                ;set screen pointer to center of screen
0016  BA 0064                           mov   dx,100d   ;# rows divided by 2
0019  B9 00A0                           mov   cx,160d   ;# columns div by 2

                                ;get character from keyboard, using ROM BIOS
                                ;  routine
001C                            get_char:
001C  B4 00                             mov   ah,0      ;code for read char
001E  CD 16                             int   16h       ;keyboard I/O ROM call
0020  3C 1B                             cmp   al,escape ;is it escape char?
0022  74 2B                             jz    exit      ;yes

0024  3C 33                             cmp   al,33h    ;is it more than "3"
0026  7F 0A                             jg    plot_it   ;yes, not a color
0028  3C 30                             cmp   al,30h    ;is it less than "0"
002A  7C 06                             jl    plot_it   ;yes, not a color

                                ;number from 0 to 3, so it is a color value
002C  24 03                             and   al,3      ;mask off upper 5 bits
002E  8A D8                             mov   bl,al     ;save it in BL
0030  EB EA                             jmp   get_char  ;get next character

                                ;figure out which way to go, and draw new line
0032                            plot_it:
0032  8A C4                             mov   al,ah     ;put scan code in AL
0034  3C 48                             cmp   al,up     ;is it UP arrow?
0036  75 01                             jnz   not_up    ;no
0038  4A                                dec   dx        ;yes, decrement row
0039                            not_up:
 0039  3C 50                             cmp   al,down   ;is it DOWN arrow?
003B  75 01                             jnz   not_down  ;no
003D  42                                inc   dx        :yes, increment row
```

```
ØØ3E                            not_down:
ØØ3E  3C 4D                             cmp   al,right  ;is it RIGHT arrow?
ØØ4Ø  75 Ø1                             jnz   not_right ;no
ØØ42  41                                inc   cx        ;yes, increment column
ØØ43                            not_right:
ØØ43  3C 4B                             cmp   al,left   ;is it LEFT arrow?
ØØ45  75 Ø1                             jnz   lite_it   ;no
ØØ47  49                                dec   cx        ;yes, decrement column

                                ;call PLOTSUB routine to write dot
                                ;requires row # in DX, col in CX, color in AL
ØØ48                            lite_it:
ØØ48  8A C3                             mov   al,bl     ;put color back in AL
ØØ4A  E8 ØØ5Ø R                         call  plotsub   ;write dot
ØØ4D  EB CD                             jmp   get_char  ;go get next arrow
ØØ4F  CB                        exit:   ret             ;return to DOS

ØØ5Ø                            main    endp    ;end of main part of program

                                ;-----------------------------------------
ØØ5Ø                            plotsub proc    near

                                ;SUBROUTINE TO PLOT A POINT ON SCREEN
                                ;       Medium res graphics mode
                                ;       32Ø x 2ØØ color
                                ;Enter with:
                                ;   X-coordinate in CX (column  number: Ø-319)
                                ;   Y-coordinate in DX (row number: Ø-199)
                                ;   color in AL (Ø=off, 1,2,3=colors)

ØØ5Ø  53                                push bx         ;save BX
ØØ51  51                                push cx         ;save column
ØØ52  52                                push dx         ;save row
ØØ53  5Ø                                push ax         ;save color

                                ;multiply the row number by # of bytes per row
                                ; (8Ø, but since already mult by 2, use 4Ø)
ØØ54  52                                push dx         ;save row for odd/even
ØØ55  BØ 28                             mov  al,4Ø      ;bytes/row div by 2
ØØ57  81 E2 ØØFE                        and  dx,Øfeh    ;mask off odd/even bit
ØØ5B  F6 E2                             mul  dl         ;AX now is row address

                                ;figure out if we should add 2ØØØh for 2nd
                                ;  memory bank, if odd row number
ØØ5D  5A                                pop  dx         ;get original row #
ØØ5E  F6 C2 Ø1                          test dl,1       ;test odd/even bit
ØØ61  74 Ø3                             jz   not_odd    ;jump on even row
ØØ63  Ø5 2ØØØ                           add  ax,2ØØØh   ;add to get 2nd bank
```

```
0066                            not_odd:
0066  8B D8                             mov  bx,ax      ;save row addr in BX

                                ;add column address to row address
0068  51                          push cx         ;save column address
0069  D1 E9                       shr  cx,1       ;shift it right to
006B  D1 E9                       shr  cx,1       ;  kill BIT-POS bits
006D  03 D9                       add  bx,cx      ;add it to addr in BX

                                ;use BIT-POS bits to put COLOR and MASK in
                                ;  the right position
006F  59                          pop  cx         ;get original col #
0070  81 E1 0003                  and  cx,3       ;save BIT-POS bits
0074  41                          inc  cx         ;get one free shift
0075  58                          pop  ax         ;get color
0076  B2 FC                       mov  dl,0fch    ;DL=mask:  11,11,11,00
0078                            shift:            ;AL=color: 00,00,00,cc
0078  D0 C8                       ror  al,1       ;shift color
007A  D0 C8                       ror  al,1       ;  two bits right
007C  D0 CA                       ror  dl,1       ;shift mask
007E  D0 CA                       ror  dl,1       ;  two bits right
0080  E2 F6                       loop shift      ;do it BIT-POS times

                                ;get contents of byte, mask off all but color
                                ;  bits, OR on color bits.

0082  26: 20 97 0000 R                and   [v_buff + bx].dl ;mask off
0087  26: 08 87 0000 R                or    [v_buff + bx].al ;OR on color

008C  5A                          pop  dx         ;restore row
008D  59                          pop  cx         ;restore column
008E  5B                          pop  bx         ;restore BX
008F  C3                          ret             ;return

0090                            plotsub endp
                                ;-----------------------------------------
0090                            pro_nam ends    ;end of code segment
                                ;*****************************************

                                        end     start   ;end assembly
```

Operating the PLOTSUB Subroutine

PLOTSUB uses the CX, DX, and AL registers to hold the column number, the row number, and the color value, respectively. Figure 10-5 shows how these registers will look.

The first thing the PLOTSUB subroutine does is translate the screen row number in DX into the memory address which is the

beginning of the row. To do this, the row number must be multiplied by the number of bytes per row, which is 80d (since there are 320 pixels per row, and 4 pixels in each byte, and 320/4=80.) However, the rightmost bit of the row number is used to specify whether the row is odd or even, so it doesn't count in the above multiplication. Its presence does mean that the rest of the number is shifted left one bit, or in effect has already been multiplied by two. So instead of multiplying by 80d, we can multiply by 40d. The result is the offset address of the beginning of the row where our pixel is located.

Next the program checks to see if the row is odd or even. If it's an odd row, then 2000h must be added to the address, since it's in the second memory bank, the one which starts at 2000h. We do this at program addresses 005D to 0066, and to do it neatly we make use of a new instruction: TEST.

The TEST Instruction

This instruction provides a convenient way of seeing if a particular bit or group of bits is set in a particular register. It performs a logical AND just like an AND instruction, but it doesn't change the operand; it merely sets the flags *as if* the AND had been carried out. In this way it's like CMP, which performs a subtraction, but also doesn't change the operand.

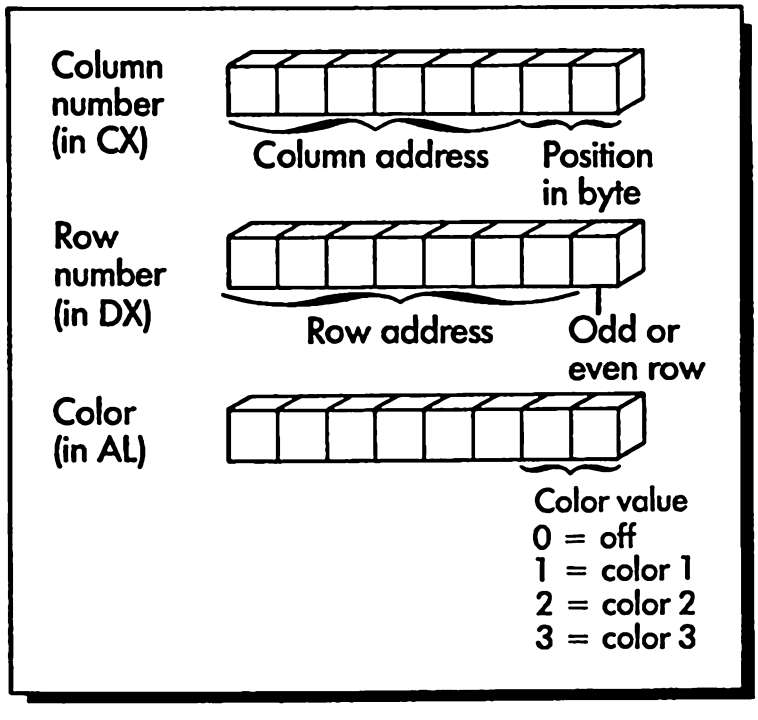


Figure 10-5. Registers used to plot a dot

TEST Instruction

ANDs two operands, but does not change them.

To test two registers:

```
TEST AL,BL
TEST BX,CX
```

To test constant and register:

```
TEST DL,Ø3h
```

To test register and memory:

```
TEST MWORD,DX
```

This instruction is most often used to see if a particular bit or bits are set in a register or memory location.

Flags affected: AF, CF, OF, PF, SF, ZF

TEST is just what we need to see if the odd/even bit is set in the row number: we TEST DL against a constant of 1, which is the bit we want to examine. (We could have tested DX instead of DL, but we don't care about the high-order bits, and testing a byte is a shorter instruction than testing a word.) If the bit is set, the the zero flag is set to NZ (not zero), and the program will add the 2000h to obtain the address in the second, odd-numbered, memory bank. This address is saved in the BX register.

The program then adds the column address to the row address already calculated in BX. Since the rightmost two bits of the column address specify which *bits* in the byte are being used, the program shifts the CX register right twice to get rid of them (temporarily) before adding the result to BX. BX now holds the address of the pixel we're going to change.

Now we have to figure out which two bits in this address are to have the new color value written into them. If these two bits are 00, then the leftmost pixel will be modified; if they are 01, then the second pixel from the left will be modified, and so on.

What we want to do is to write the two new color value bits into this position in the byte, while leaving the other six bits undisturbed. So we want to AND off the two bits in question in the original byte, then OR on the two new color bits. However, the location of the color bits, and the mask used to AND off the old color bits, will be in one of four different

places, depending on the value of the bit position bits. Thus we must generate new configurations for both these quantities. The program does this by starting with both the color bits and the mask in the rightmost two bit positions, and rotating them once if BIT-POS is 00, twice if BIT-POS is 01, and so on. The rotations are carried out in locations 0078 to 0080. Figure 10-6 shows the relationship of the color bits and the mask to the bit-position values.

That's about it. The program then ANDs off the unwanted two bits in the byte in the memory location in BX, and ORs on the color bits, as described above. And presto, a dot appears (or disappears) on your color screen.

Drawing Lines

Writing a program to draw straight lines on the screen is pretty straightforward, so long as the lines fall into one of two categories: vertical or horizontal. We're going to show you a short program that draws a grid of lines on the screen, like a checkerboard, to show you how simple it is to draw lines that make an angle of either 0 degrees or 90 degrees from the horizontal. Drawing lines that make any *other* angle, on the other hand, is surprisingly difficult. We'll cover that in the last section of this chapter.

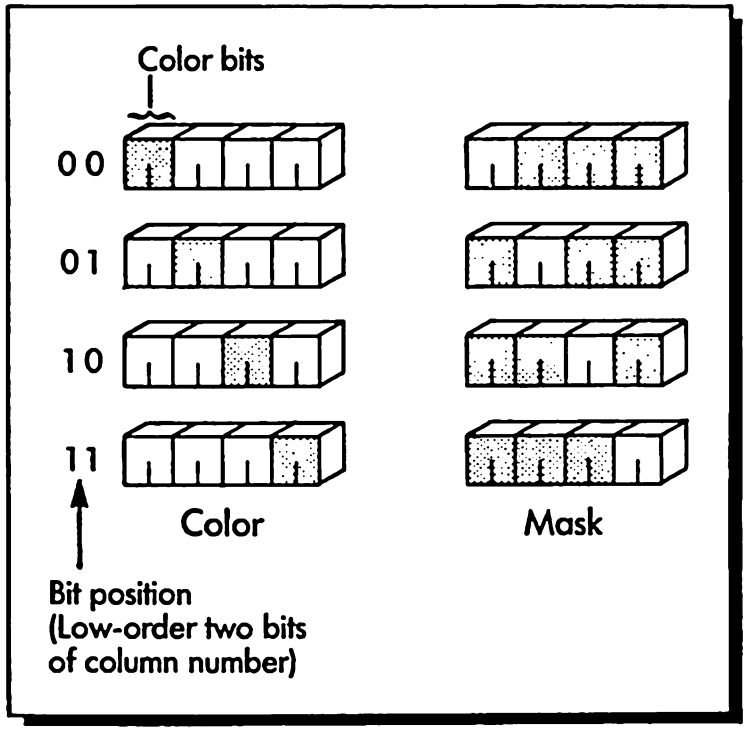


Figure 10-6. Bit positions in memory addresses

The GRID Program: Vertical and Horizontal Lines

The program which follows draws vertical lines every 20d pixels across the screen, and horizontal lines every 20d pixels down the screen. The result is a grid-like checkerboard pattern.

```
                        ;GRID--Program to draw grid on screen
                        ;       Uses ROM routine

                        ;For 32Ø x 2ØØ medium res color mode

                        ;*******************************************

ØØØØ                    pro_nam segment         ;define code segment

                        ;-------------------------------------------
ØØØØ                    main    proc    far     ;main part of program

                                assume  cs:pro_nam

ØØØØ                    start:                  ;starting execution address

                        ;set up stack for return
ØØØØ  1E                       push    ds      ;save DS
ØØØ1  2B CØ                    sub     ax,ax   ;set AX to zero
ØØØ3  5Ø                       push    ax      ;put it on stack

                        ;clear screen by scrolling it, using ROM call

ØØØ4  B4 Ø6                    mov  ah,6       ;scroll up function
ØØØ6  BØ ØØ                    mov  al,Ø       ;code to blank screen
ØØØ8  B9 ØØØØ                  mov  cx,Ø       ;upper left = Ø,Ø
ØØØB  B2 4F                    mov  dl,79      ;lower right corner
ØØØD  B6 18                    mov  dh,24      ;  at 79,24
ØØØF  CD 1Ø                    int  1Øh        ;call video interrupt

                        ;pixel location kept in CX, DX registers
                        ;   row    number (Ø to 2ØØd) in DX
                        ;   column number (Ø to 32Ød) in CX

                        ;DRAW HORIZONTAL LINES EVERY 2Ø PIXELS
Ø11  BA ØØØØ                   mov  dx,Ø       ;set to first line

                        ;draw one horizontal line at DX
ØØ14                    hline:
ØØ14  B9 ØØØØ                  mov  cx,Ø       :start of horiz line
ØØ17                    hdot:
ØØ17  BØ Ø1                    mov  al,1       ;set color to 1
ØØ19  B4 ØC                    mov  ah,12d     :write dot function #
```

```
ØØ1B  CD 1Ø                     int   1Øh       ;call Video ROM
ØØ1D  41                       inc   cx        ;next dot
ØØ1E  81 F9 Ø12C                cmp   cx,3ØØ    ;done all dots?
ØØ22  7C F3                     jl    hdot      ;not yet

                        ;next horizontal line
ØØ24  83 C2 14                  add   dx,2Ø     ;advance to next line
ØØ27  81 FA ØØC8                cmp   dx,2ØØ    ;off the screen yet?
ØØ2B  7C E7                     jl    hline     ;not yet

                        ;DRAW VERTICAL LINES EVERY 2Ø PIXELS
ØØ2D  B9 ØØØØ                   mov   cx,Ø      ;set to first line

                        ;draw one vertical line at CX
ØØ3Ø                    vline:
ØØ3Ø  BA ØØØØ                   mov   dx,Ø      ;start of vert line
ØØ33                    vdot:
ØØ33  BØ Ø2                     mov   al,2      ;set color to 2
ØØ35  B4 ØC                     mov   ah,12d    ;write dot function #
ØØ37  CD 1Ø                     int   1Øh       ;call Video ROM
ØØ39  42                       inc   dx        ;next dot
ØØ3A  81 FA ØØB4                cmp   dx,18Ø    ;done all dots?
ØØ3E  7C F3                     jl    vdot      ;not yet

                         ;next vertical line
ØØ4Ø  83 C1 14                  add   cx,2Ø     ;advance to next line
ØØ43  81 F9 Ø14Ø                cmp   cx,32Ø    ;off the screen yet?
ØØ47  7C E7                     jl    vline     ;do next line

ØØ49  CB                       ret             ;return to DOS

ØØ4A                    main    endp    ;end of main part of program
                        ;-----------------------------------------
ØØ4A                    pro_nam ends    ;end of code segment
                        ;*****************************************

                                end     start   ;end assembly
```

The GRID program works by starting off with row and column numbers of zero. DX and CX are used to hold the row and column numbers as usual. To produce each horizontal line, DX is held fixed, while CX is incremented one pixel at a time, from 0 to the end of the line at 300. Likewise, to produce each vertical line, CX is held constant, while DX is incremented pixel by pixel until the end of the line at 100. For simplicity, the ROM BIOS video Write Dot routine is used to put the dots on the screen.

Diagonal Lines

Now we come to the interesting part of the story: drawing lines which *aren't* vertical or horizontal. For example, suppose we want to draw a line which rises 3 pixels vertically while it's going across the screen 5 pixels. At the end points (0,0) and (5,3) the line passes exactly through the center of the dot locations on the video screen. But between the end points the line does not fall on any of the possible pixel positions. Figure 10-7 shows what this looks like.

The question is, then, how is a line-drawing program going to know where to put the intermediate pixels? Since it can't put them exactly on the line, it has to make a decision about which of two possible pixel positions to put each dot on. The dashed lines between pixel locations in the figure connect the possible choices at each location.

Multiplying by the Slope

If you remember your geometry, you may recall that any diagonal line can be represented by the formula:

$$y = mx + b$$

where m is the slope of the line, and b is a constant which determines at what point the line will pass through the Y-axis. In Figure 10-7, b is zero, and the slope of the line is 3/5, so we have:

$$y = 3/5 * x$$

Now, when we're drawing the points on the line, it's easy to change

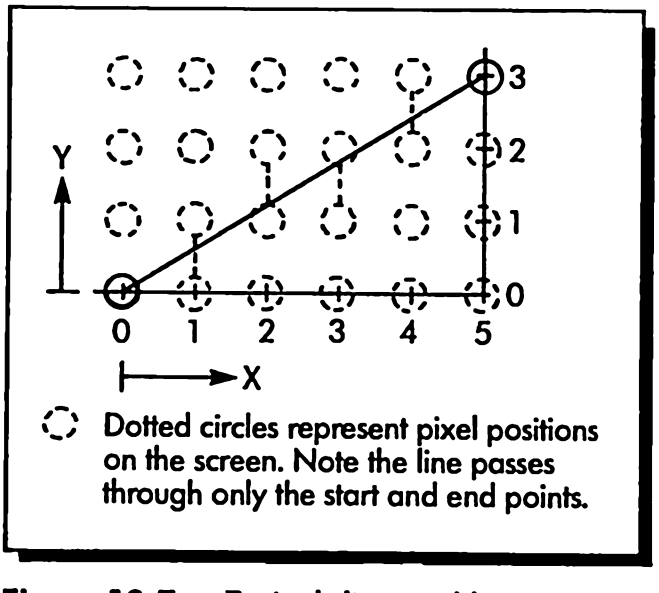


Figure 10-7. Typical diagonal line

the X coordinate; we simply add one to it, so X goes from 0 to 1 to 2 up to the end point at 5. But it's not so easy to figure out the corresponding Y coordinates. When X is 0, then Y is 0. What's Y when X is 1? Applying the formula above,

$$y = 3/5 * 1 = 3/5$$

Similarly, when X is 2,

$$y = 3/5 * 2 = 6/5$$

Solving the equation this way tells us where Y is *supposed* to be. However, since we can only plot pixels on integer values of Y, we have to round off the Y values: 3/5 rounds off to 1, as does 6/5. The next Y value is 9/5, which rounds off to 2, and so on. Figure 10-8 shows how the pixels look when drawn using this algorithm.

So what's wrong with this system? The problem is that it takes too long to calculate points this way. There are often hundreds of points on a line (not six, as in our example), and in many graphics programs there are a lot of lines, or even lines that need to appear to move (as for instance when you're landing a plane on a simulated aircraft-carrier deck, and the deck must get closer to you in real time). So routines to draw lines must be *fast*.

There are two reasons why multiplying by the slope is slow. First, multiplication takes a long time. Even though the 8088 in the IBM PC can multiply in one instruction (unlike many computers), the instruction itself is a slow one (relative to other instructions such as ADD). Second, because the slope is a fraction, we can't use simple integer arithmetic. We must use *real* numbers (single precision), or *binary fractions*, both of which are much slower and more complex to operate on. What we need is a different approach altogether.

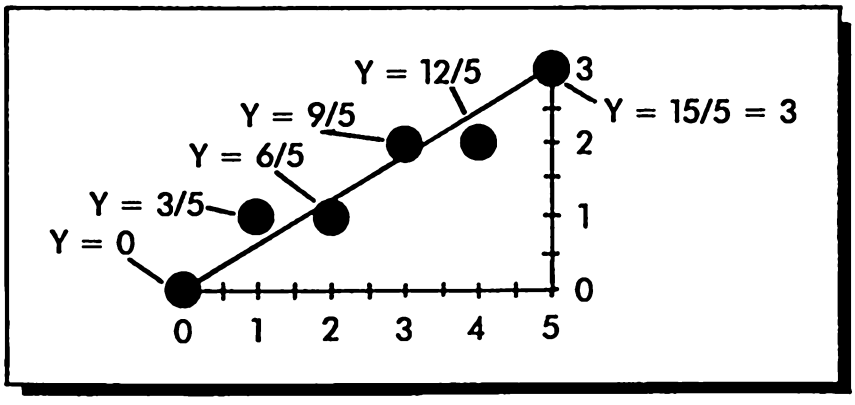


Figure 10-8. Calculating points using the slope

Bresenham's Algorithm

Along comes Bresenham, with a line-drawing method which has become famous in the computer industry. Essentially his method involves keeping track of a number, called an *error term*, which is related to the difference between where the pixel *should* go, if it could be drawn right on the line, and where it *must* go, since it can only occupy integer pixel locations. Each time we move over one unit in the X direction, we add a certain constant to this error term. If the resulting new value for the error term is big enough, we increment Y (so the pixel at the next higher value of X is plotted one point higher on the Y-axis than before), and

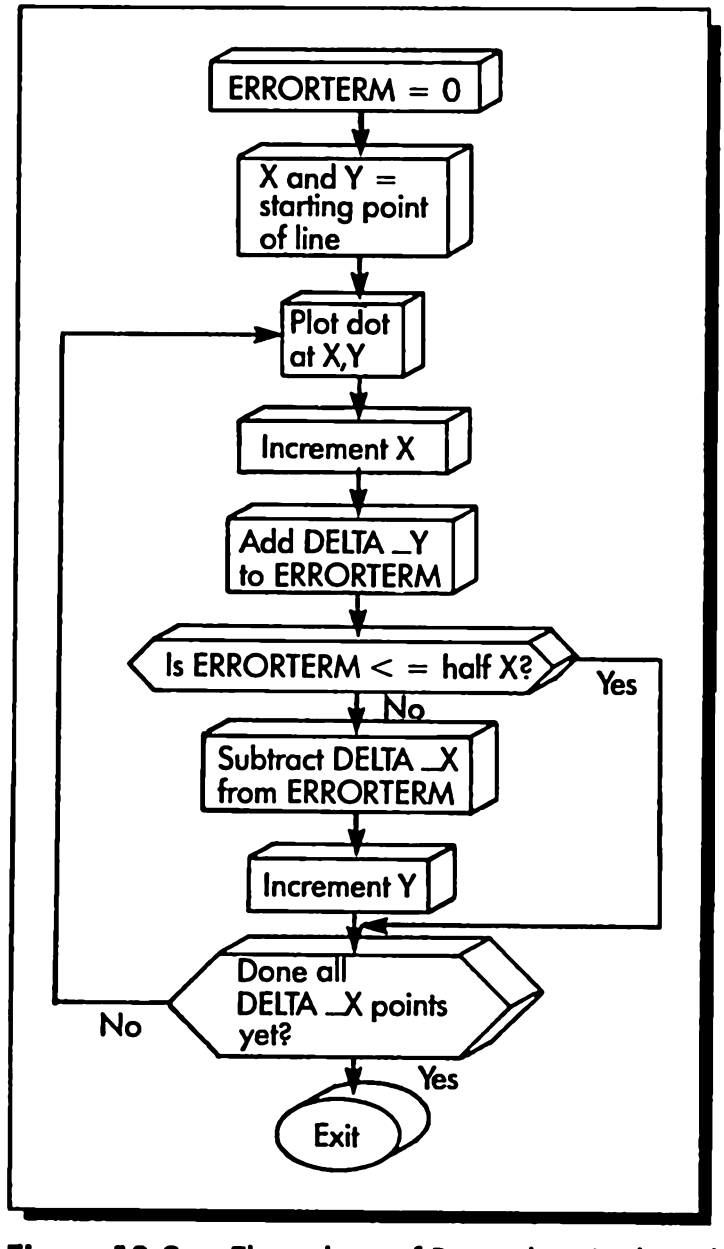


Figure 10-9. Flow chart of Bresenham's algorithm

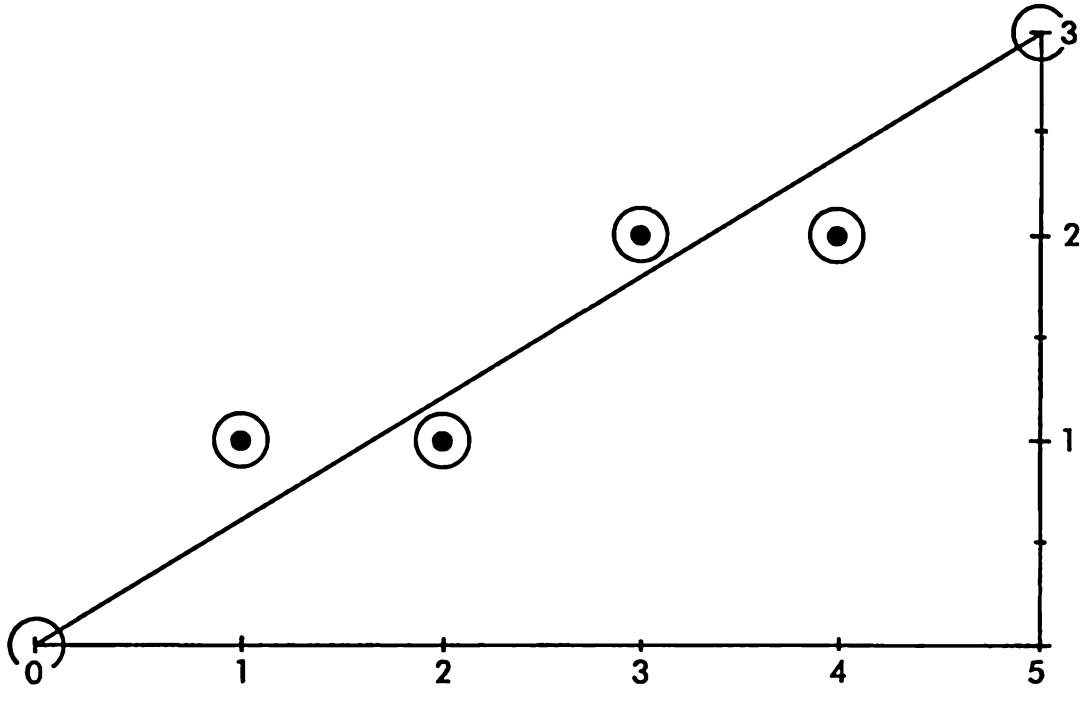
also subtract another constant from the error term. Sound confusing? It'll be clearer when we run through a specific example.

Let's define some terms, so we can talk about things more clearly. X and Y will be the coordinates where we're going to plot a particular pixel. In Figure 10-8, X and Y both start at 0; X ends at 5 and Y ends at 3. The difference between the starting and ending addresses we'll call delta_x and delta_y (delta stands for difference). So

delta_x = 5 – 0 = 5, and

delta_y = 3 – 0 = 3

We'll also define half of delta_x as a constant called halfx. However,



ERRORTERM	0	–2	1	–1	2
Add 3 (DELTA_Y)	3	1	4	2	
Result <=2?	No	Yes	No	Yes	No
Subtract 5 (DELTA_X)	–2		–1		0
Increment Y?	Yes	No	Yes	No	Yes

Figure 10-10. Applying the algorithm to a 5-over by 3-up line

halfx must be an integer, so we truncate any fractional part. Thus

halfx = delta_x / 2 = 5 / 2 = 2

Figure 10-9 shows a flow chart of Bresenham's algorithm, using these definitions.

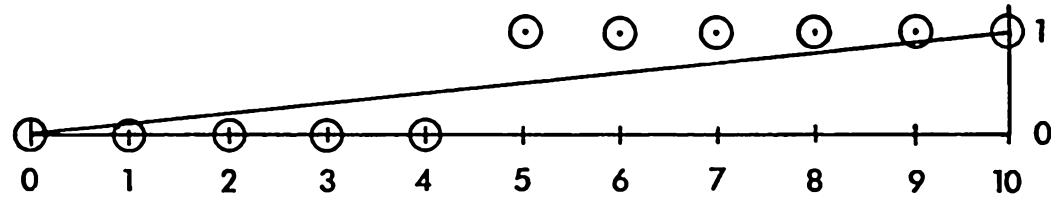
Let's see what happens when we apply the algorithm to our original example. X and Y both start at 0, as does the error term. As we saw, delta_x is 5, delta_y is 3, and halfx is 2. Figure 10-10 shows how the error term changes as we apply the algorithm to it. By comparing this figure with the flow chart in Figure 10-9, you should be able to follow the use of the algorithm.

We start with the error term at zero, and plot the point at the current values of X and Y: 0,0. X is then automatically incremented. To figure out whether to increment Y, we first add delta_y, which is 3, to the error term. If the result is less than half_x, which is 2, no action is taken. However, if the result is greater than half_x, then Y in incremented and delta_x, which is 5, is subtracted from the error term.

The process is repeated for subsequent points until X reaches the end of the line.

Making the Steps Symmetrical

One of the desirable attributes of a good line-drawing algorithm is that the steps in the line be symmetrical. This is especially important in



ERRORTERM	0	1	2	3	4	−5	−4	−3	−2	−1
Add 1	1	2	3	4	5	−4	−3	−2	−1	0
Result < = 5 ?	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
Subtract 10					−5					
Increment Y?	No	No	No	No	Yes	No	No	No	No	No

Figure 10-11. Applying the algorithm to a 10-over by 1-up line

lines which are almost horizontal or almost vertical. For instance, suppose we want to draw a line from 0,0 to 10,1. That is, the line will go 10 units in the horizontal direction, while it's rising only 1 unit in the vertical direction. At some point in its length the line will have to change from Y=0 to Y=1; ideally we would like this to happen in the middle of the line, where X=5.

Figure 10-11 shows Bresenham's algorithm applied to this situation. Here, delta_x = 10, delta_y = 1, and halfx = 5. Note how the algorithm causes the line to break in the middle, just where we want.

The DRAWLINE Program

The program DRAWLINE, shown in the listing below, demonstrates the use of Bresenham's algorithm on the 320x200 color screen. The heart of the program is the LINESUB routine, which actually draws the line. This routine can be used in any program requiring rapid line drawing. You can incorporate it in programs which draw stars or cat's cradle patterns, or if you're ambitious, you can get started on the carrier-landing simulation program we mentioned earlier.

To use DRAWLINE, type in five decimal numbers: X1, Y1 (the starting coordinates of the line); X2, Y2 (the ending coordinates); and a color value from 0 to 3. After the fifth number the program will draw the line and then wait for the next set of five numbers to be typed in. To return to DOS, press the Esc key. The program should be run as an EXE file, and CHAMODE should be used (if necessary) to shift the system into the 320x200 color mode before running the program.

```
                        ;DRAWLINE--Program to draw diagonal lines
                        ;          Uses ROM routines

                        ;For 32Øx2ØØ color graphics modes

                        ;*******************************************

ØØØØ                    datarea segment         ;define data segment

ØØØØ  ????              delta_x dw      ?       ; |x2-x1|
ØØØ2  ????              delta_y dw      ?       ; |y2-y1|
ØØØ4                    halfy   label   word    ; |y2-y1| / 2
ØØØ4  ????              halfx   dw      ?       ; |x2-x1| / 2
ØØØ6  ????              count   dw      ?       ; set to long axis

ØØØ8  ????              x1      dw      ?       ; first X coordinate
ØØØA  ????              y1      dw      ?       ; first Y coordinate
```

```
000C  ????                    x2      dw      ?       ; second X coordinate
000E  ????                    y2      dw      ?       ; second Y coordinate
0010  ????                    color   dw      ?       : color, 0-1 or 0-4

0012                          datarea ends
                              ;*****************************************

0000                          pro_nam segment         ;define code segment

                              ;-----------------------------------------
0000                          main    proc    far     ;main part of program

                                      assume  cs:pro_nam,ds:datarea

0000                          start:                  ;starting execution address

                              ;set up stack for return
0000  1E                              push    ds      ;save DS
0001  2B C0                           sub     ax,ax   ;set AX to zero
0003  50                              push    ax      ;put it on stack

                              ;clear screen by scrolling it, using ROM call

0004  B4 06                           mov  ah,6       ;scroll up function
0006  B0 00                           mov  al,0       ;code to blank screen
0008  B9 0000                         mov  cx,0       ;upper left = 0,0
000B  B2 4F                           mov  dl,79      :lower right corner
000D  B6 18                           mov  dh,24      ;  at 79,24
000F  CD 10                           int  10h        ;call video interrupt

                              ;SET UP BEGINNING AND END OF LINE
                              :  AND CALL LINESUB TO DRAW LINE

0011                          newline:
0011  E8 00F2 R                       call decibin    ;x1
0014  89 1E 0008 R                    mov  x1,bx
0018  E8 00F2 R                       call decibin    ;y1
001B  89 1E 000A R                    mov  y1,bx
001F  E8 00F2 R                       call decibin    ;x2
0022  89 1E 000C R                    mov  x2,bx
0026  E8 00F2 R                       call decibin    ;y2
0029  89 1E 000E R                    mov  y2,bx
002D  E8 00F2 R                       call decibin    ;color
0030  89 1E 0010 R                    mov  color,bx

0034  E8 0039 R                       call linesub    ;draw line

0037  EB D8                           jmp  newline    :do it again
```

```
0039                    main    endp    ;end of main part of program
                        ;-----------------------------------------
0039                    linesub proc    near

                        ;LINESUB--SUBROUTINE TO DRAW LINE
                        ;
                        ;Input is x1, y1  (start of line)
                        ;         x2, y2  (end of line)
                        ;         color (0-1 or 0-4)


                        ;find |y2-y1| -- result is delta_y
0039  A1 000E R                 mov  ax,y2        ;get y2
003C  2B 06 000A R              sub  ax,y1        ;subtract y1
                                                  ;result in AX
                        ;figure out if delta_y is positive or negative
                        ;  SI=1 if positive, SI=-1 if negative
0040  BE 0001                  mov  si,1       ;set flag to positive
0043  7D 05                     jge  store_y    ;  keep it that way
0045  BE FFFF                  mov  si,-1      ;set flag to negative
0048  F7 D8                    neg  ax         ;set to abs value
004A                    store_y:
004A  A3 0002 R                 mov  delta_y,ax ;store delta_y

                        ;find |x2-x1| -- result is delta_x
004D  A1 000C R                 mov  ax,x2      ;get x2
0050  2B 06 0008 R              sub  ax,x1      ;subtract x1
                                                ;result in AX
                        ;figure out if delta_x is positive or negative
                        ;  DI=0 if positive, DI=1 if negative
0054  BF 0001                   mov  di,1       ;set flag to positive
0057  7D 05                     jge  store_x    ;  keep it that way
0059  BF FFFF                  mov  di,-1      ;set flag to negative
005C  F7 D8                    neg  ax         ;set to abs value
005E                    store_x:
005E  A3 0000 R                 mov  delta_x,ax ;store delta_x

                         ;figure out if slope is greater or less than 1
0061  A1 0000 R                 mov  ax,delta_x ;get delta_x
0064  3B 06 0002 R              cmp  ax,delta_y ;compare deltas
0068  7C 06                     jl   csteep     ;slope > 1
006A  E8 0074 R                 call easy       ;slope < 1, or = 1
006D  EB 04 90                  jmp  finish

0070                    csteep:
0070  E8 00A9 R                 call steep      ;slope > 1
```

```
                                ;DONE LINE--RETURN
0073                            finish:
0073  C3                                ret

0074                            linesub endp
                                ;-----------------------------------------
0074                            easy    proc  near

                                ;SLOPE < 1

                                ;calculate half of delta_x, call it halfx
0074  A1 0000 R                         mov  ax,delta_x ;get |x2-x1|
0077  D1 E8                             shr  ax,1       ;shift right to divide
0079  A3 0004 R                         mov  halfx,ax   ;  by 2

                                ;initialize values

007C  8B 0E 0008 R                      mov  cx,x1      ;set x1
0080  8B 16 000A R                      mov  dx,y1      ;set y1
0084  BB 0000                           mov  bx,0       ;initialize BX
0087  A1 0000 R                         mov  ax,delta_x ;set count
008A  A3 0006 R                         mov  count,ax   ;  to |x2-x1|
008D                            newdot:
008D  E8 00DE R                         call dotplot    ;plot the dot
0090  03 CF                             add  cx,di      ;inc/dec X
0092  03 1E 0002 R                      add  bx,delta_y ;add |y2-y1| to BX
0096  3B 1E 0004 R                      cmp  bx,halfx   ;compare to |x2-x1|/2
009A  7E 06                             jle  dcount     ;  (don't inc/dec Y)
009C  2B 1E 0000 R                      sub  bx,delta_x ;subtract |x2-x1|
                                                        ;  from BX
00A0  03 D6                             add  dx,si      ;inc/dec Y
00A2                            dcount:
00A2  FF 0E 0006 R                      dec  count      ;done line yet?
00A6  7D E5                             jge  newdot     ;not yet

00A8  C3                                ret             ;done line

00A9                            easy    endp
                                ;-----------------------------------------
00A9                            steep   proc  near

                                ;SLOPE > 1

                                ;calculate half of delta_y, call it halfy
00A9  A1 0002 R                         mov  ax,delta_y ;get |y2-y1|
00AC  D1 E8                             shr  ax,1       ;shift right to divide
00AE  A3 0004 R                         mov  halfy,ax   :  by 2
```

```
                                ;initialize values
00B1  8B 0E 0008 R                      mov  cx,x1      ;set x1
00B5  8B 16 000A R                      mov  dx,y1      ;set y1

00B9  BB 0000                           mov  bx,0       ;initialize BX
00BC  A1 0002 R                         mov  ax,delta_y ;set count
00BF  A3 0006 R                         mov  count,ax   ;  to x2-y1

00C2                            newdot2:
00C2  E8 00DE R                         call dotplot    ;plot the dot
00C5  03 D6                             add  dx,si      ;inc/dec Y
00C7  03 1E 0000 R                      add  bx,delta_x ;add |x2-x1| to BX
00CB  3B 1E 0004 R                      cmp  bx,halfy   ;compare to |y2-y1|/2
00CF  7E 06                             jle  dcount2    ;don't inc/dec X
00D1  2B 1E 0002 R                      sub  bx,delta_y ;subtract |y2-y1|
                                                        ;  from BX
00D5  03 CF                             add  cx,di      ;inc/dec X
00D7                            dcount2:
00D7  FF 0E 0006 R                      dec  count      ;done line yet?
00DB  7D E5                             jge  newdot2    ;not yet

00DD  C3                                ret             ;return to main dline

00DE                            steep   endp
                                ;-----------------------------------------
00DE                            dotplot proc near

                                ;SAVE REGISTERS AND CALL PLOT ROUTINE

00DE  53                                push bx         ;save registers
00DF  51                                push cx
00E0  52                                push dx
00E1  50                                push ax
00E2  56                                push si
00E3  57                                push di

                                ;use ROM routine to write dot
                                ;requires row # in DX, col in CX, color in AL

00E4  A1 0010 R                         mov   ax,color  ;set color value
00E7  B4 0C                             mov   ah,12d    ;write dot function
00E9  CD 10                             int   10h       ;video BIOS routine

00EB  5F                                pop  di         ;restore registers
00EC  5E                                pop  si
00ED  58                                pop  ax
00EE  5A                                pop  dx
00EF  59                                pop  cx
00F0  5B                                pop  bx
```

```
00F1  C3                        ret             ;return

00F2                    dotplot endp
                        ;-----------------------------------------
                        ;
= 0001                  key_in  equ     1h      ;keyboard input
= 0021                  doscall equ     21h     ;DOS interrupt number
                        ;
00F2                    decibin proc    near
                        ;
                        ;SUBROUTINE TO CONVERT DEC ON KEYBD TO BINARY
                        ;  result is left in BX register
                        ;
00F2  BB 0000                   mov     bx,0    ;clear BX for number
                        ;
                        ;get digit from keyboard, convert to binary
00F5                    newchar:
00F5  B4 01                     mov   ah,key_in ;keyboard input
00F7  CD 21                     int     doscall ;call DOS
00F9  2C 30                     sub     al,30h  ;ASCII to binary
00FB  7C 10                     jl      exit    ;jump if < 0
00FD  3C 09                     cmp     al,9d   ;is it > 9d ?
00FF  7F 0C                     jg      exit    ;yes, not dec digit
0101  98                        cbw     ;byte in AL to word in AX
                        ;(digit is in AX)
                        ;
                        ;multiply number in bx by 10 decimal
0102  93                        xchg    ax,bx   ;trade digit & number
0103  B9 000A                   mov     cx,10d  ;put 10 dec in CX
0106  F7 E1                     mul     cx      ;number times 10
0108  93                        xchg    ax,bx   ;trade number & digit
                        ;
                        ;add digit in ax to number in bx
0109  03 D8                     add     bx,ax   ;add digit to number
010B  EB E8                     jmp     newchar ;get next digit
010D                    exit:
010D  C3                        ret
                        ;
010E                    decibin endp
                        ;-----------------------------------------
010E                    pro_nam ends    ;end of code segment
                        ;*****************************************

                                end     start   ;end assembly
```

The real work in the LINESUB routine is performed in the following section of code:

```
        mov  cx,x1      ;set x1
        mov  dx,y1      ;set y1
        mov  bx,Ø       ;initialize BX
        mov  ax,delta_x ;set count
        mov  count,ax   ;  to |x2-x1|
newdot:
        call dotplot    ;plot the dot
        add  cx,di      ;inc/dec X
        add  bx,delta_y ;add |y2-y1| to BX
        cmp  bx,halfx   ;compare to |x2-x1|/2
        jle  dcount     ;  (don't inc/dec Y)
        sub  bx,delta_x ;subtract |x2-x1|
                        ;  from BX
        add  dx,si      ;inc/dec Y
dcount:
        dec  count      ;done line yet?
        jge  newdot     ;not yet
```

The first five instructions here initialize various values. Then, starting at "newdot," the line is actually drawn. This section of code corresponds with the flow chart shown in Figure 10-9. By comparing the code and the flow chart, you should have no trouble understanding how it works. There are only a few short instructions in this loop, so the line can be plotted very quickly.

There are several complexities in the LINESUB routine which we haven't dealt with yet.

Complexity One

The first complexity is that the end points of the line may be such that X1 is greater than X2, or Y1 is greater than Y2. When this is the case we need to *decrement* the values of X or Y as we draw the line, rather than increment them. For this reason, the program figures out the sign of X2 – X1 and sets the DI register to either +1 or –1 accordingly. Similarly, SI is set to +1 or –1 according to the sign of Y2 – Y1. DI and SI are then added to the X and Y values in CX and DX to effect the appropriate increment/decrement.

Complexity Two

The second complexity is that the slope of the line may be less than 1 or greater than 1. (A slope of 1 corresponds to a 45-degree angle.) So far we've examined cases where the slope is less than 1. In this case Y changes less than X over the course of the line, so we know we *always* increment X to get to the next point, and that we *sometimes* increment Y and sometimes don't, depending on the results of the algorithm.

When the slope of the line is greater than 1, the situation is reversed. We need to increment Y every time, and sometimes increment X and sometimes not. Otherwise there won't be enough pixels drawn on the line. (For instance, a line that went from X=0 to 3, and from Y=0 to 10, would have only 3 dots on it if we were to draw only one dot for every X position.)

Thus we need two parts of the program: one for slopes less than 1 and another for slopes greater than 1. The two procedures EASY and STEEP are used for these two cases.

There are many refinements that can be made to this program to make the generation of lines even faster. However, most of these refinements make the program more difficult to understand. For instance, a possible enhancement is to use self-modifying code. This gives some increase in speed, but makes the program harder to understand, and harder to write and debug. We won't pursue these esoteric matters further here.

Summary

In this chapter we've covered some of the more common techniques used in graphics. You should have an idea how the bytes in memory correspond to the pixels on the screen in both the monochrome display and the 320x200 color display. You've learned how to plot points on the screen, and how to draw vertical, horizontal, and even diagonal lines. This should be all you need to get started in the space-age field of computer graphics!

11

Reading and Writing Disk Files

Concepts

File access — Sequential, Random, and Random Block
File Control Block (FCB)
Data Transfer Area (DTA)

DOS Functions

Open File
Sequential Read
Create File
Sequential Write
Close File
Random Read
Random Block Read
File Size

Applications

SET-BD — Creates a file of birthdays
GET-BD — Finds all birthdays on a given date
MOD-BD — Modifies existing birthday file
SAVEIMAG — Dumps screen image to disk file
(See appendix B for programs.)

In this chapter we explore one of the most important groups of DOS function calls: those that access the disk. Almost all but the most minor programs require disk access. If you're writing a word processing program, you need to read and write the text files from the disk. If you're writing a finance program, then you'll need to access tables of figures stored on the disk. Graphics programs will read pictures stored as

files on the disk, and so on. In short, without disk capability, few programs can do useful work.

Essentially what this chapter teaches is how to read files from the disk, and write them to the disk, in a variety of different ways. If you have been impressed before with how much trouble and effort DOS function calls can save, here you will be amazed. Instead of having to deal with bytes, tracks, sectors, rotation speed, track interleaving, and other esoteric topics, we can deal simply with filenames. The DOS disk function calls will take care of transforming our request for a file into the detailed instructions that will access the actual bits at individual locations on the disk.

This chapter will cover some of the fundamental concepts involved in disk access, and then go on to explore in more detail three of the techniques used in the IBM PC-DOS functions: sequential access, random access, and random block access. We'll save the fourth method of disk access — file handles — for the next chapter.

The Historical Perspective

The DOS functions in IBM PC-DOS versions 1.00 and 1.10 provide for three different ways to read and write files to the disk; DOS version 2.00 provides *four* different techniques. The reason there are so many different ways to do things is largely historical.

In the beginning there was CP/M (Control Program for Microprocessors), the operating system that is the distant ancestor of PC-DOS. The earliest versions of CP/M used *sequential access* to read and write files to the disk. Briefly, this meant that a file was divided into *records*, which could then be accessed only in sequence, from the first record in the file to the last record. (A record is simply a subdivision of a file, much as a book is divided into chapters.) Later versions of CP/M added a refinement to this, called *random access*. This meant that you could get at a record in the middle of a file.

To maintain a measure of compatibility with this earlier operating system, Microsoft, Inc. — the designers of PC-DOS — included both sequential and random access DOS functions in version 1 of the operating system. However, they added another method, called *random block access*. Where the earlier methods could read or write only one *record* with one function call, the random block method could read or write an entire *file* with a single call.

In PC-DOS version 2.00, Microsoft added yet another way to access disk files: with *file handles*. The need for this additional system resulted

from the introduction of tree-structured directories. In a complex directory system, a simple filename is no longer adequate to specify a file: an entire *pathname* may have to be used, such as:

```
\DIR1\LETTERS\XCORP2.TXT.
```

None of the earlier DOS functions could handle pathnames, so an entirely new system had to be introduced. Once the pathname for a file has been given to the operating system, it assigns it a *file handle*, which is a 16-bit number. From there on this simple number may be used to reference the file for reading and writing, rather than the long pathname.

Floppies and the Fixed Disk

The techniques described in this chapter apply equally well to reading and writing files on floppy diskettes and on fixed disks. As far as your programs are concerned, the fixed disk is almost the same as a diskette. One difference is that the fixed disk is always assumed to be drive C. (It can also be drive D, but let's ignore that possibility in this discussion.) To access a file (in the root directory) on the hard disk you would have to precede it by the drive specifier (C:). For example,

```
C:CHAP-11.TXT.
```

On the other hand, floppies can be in drive A or drive B — in the examples in this chapter we generally assume drive A, so that no drive specifier is needed preceding the filename.

The other difference is that it's highly probable that if you have the fixed disk you will be using the tree-structured directory (available under DOS version 2.00). If so, you will be referring to files by *pathnames*, and not simply by filenames. In this case you will have to use the file handle approach to accessing those files which are on the disk. The sequential, random, and random block access techniques will not work unless you are already in the same directory as the file you want to access (then a pathname is not required).

Even if you have a fixed disk, you can still access those files that are on floppy diskettes (in drive A or B) by using any of the methods described in this chapter.

Which System Should You Use?

Which of the four disk access systems should you use in a given

program? There's no easy answer. IBM recommends that you use the newest possible system provided with your particular operating system. That is, if you're using DOS 1.1, you should use random block access, and if you're using DOS 2.0, you should use file handles. However, there is a disadvantage to following this advice, which can be summarized in one word: compatibility. If you use file handles, your programs *will not run* on DOS versions 1.00 and 1.10. If you use random block access, your programs will run under PC-DOS versions 1 *and* 2, but it will be more difficult to translate your programs to run under CP/M, should you ever want to.

The later function calls are more powerful, but they are also, in some ways at least, more complicated. Of course, if you want to read and write to the fixed disk using pathnames, then you *must* use the file handle approach described in the next chapter.

For the time being, don't worry about which approach to disk access is the best to use. When you've finished this chapter you'll be in a better position to decide which is best in your particular situation.

What We're Going to Cover

As we've said, in this chapter we're going to introduce you to three of the file accessing systems: sequential, random, and random block. However, because of the large number of disk-related DOS functions, and the enormous amount of material that would be necessary to completely describe all the variations of each approach, we're going to cover the different methods in different depths.

We'll start by going into considerable detail about sequential access. This is the simplest method of accessing the disk, and will serve to introduce you to some of the key features of all file-accessing systems: opening and closing files, transferring information *about* the file between the disk and the operating system, and transferring the file itself.

Random access is very similar to sequential access, so we'll cover it more rapidly, but still with enough detail to get you off and running. Similarly, we'll cover random block access fairly quickly. In the next chapter we'll go into more detail when we introduce file handles.

While we don't describe every DOS function used for disk access, the material in this chapter should provide you with the ability to read, understand, and put to use almost everything you find in appendix D of the *IBM Personal Computer Disk Operating System* manual.

Some Common Concepts Behind Disk Access Techniques

No matter which approach you use to access the disk, there are

certain common requirements. First, there must be a way for your program to tell the operating system *which file to access.* This is accomplished through an area in memory where your program stores the name of the file. Sometimes data besides the filename must go in this area as well, such as which record of a file to read.

Second, there must be a place in memory where the *data* which will be read or written to the disk is stored (by the disk, when reading; and by your program, when writing). In the IBM PC this part of memory is called the Data Transfer Area, or DTA.

Third, the file must be *opened.* This is somewhat analogous to opening a file folder before you put a report in your file cabinet (or take the report out). You can't just throw the report into the cabinet at random, it must go in a particular place. By opening a file, we tell the operating system where data which we subsequently read (or write) is going to go (or come from).

> Before reading or writing to a file, the file must be opened.

Fourth, after the file has been accessed, especially if it has been written to, it must be *closed.* This is like returning the file folder to the filing cabinet after you have placed the report in it. By closing the file we make sure that the operating system knows where all the parts of the file are. If we write to a file and forget to close it, some or all of it may be lost.

The remainder of this chapter will show how these four concepts are applied to the reading and writing of data from and to the disk, using the sequential, random, and random block approaches.

Sequential Access

Sequential access is the simplest, and historically the first, method used to access material on the disk. We'll explore in detail how sequential access is used for reading and writing files to the disk.

Using DEBUG to Open a Sequential File

Let's plunge right in and *open a file.* As we mentioned, it's necessary to open a file before we can read or write to it. Opening the file alerts the operating system that we intend to access a particular file, and provides our program with some necessary information about the file. Since the

instructions to open a sequential file are so simple, we'll use DEBUG, which will then permit us to quickly explore just what happens in the computer's memory when a file is opened.

OPEN FILE Function — Number 0Fh

Enter with:

Reg AH = 0Fh
Reg DS = segment address of FCB
Reg DX = offset address of FCB

The filename and extension must be entered in the File Control Block.

Execute:

```
INT 21
```

Return with:

Reg AL = 00 if file is found
= FFh if file not found

In the File Control Block, the drive number, current block, record size, file size, and date are filled in.

Before you can open a file, it must already exist on the disk; that is, you can't open a nonexistent file. Let's create a file which we can then open. Use your word processor (or EDLIN — this is one place where EDLIN does the job just fine) to create a file consisting of several short lines of prose. Call this file TESTFILE.TXT, and store it on your disk. Let's say it looks like this:

```
Now is the time
for all good men
to come to the aid
of their country.
```

Now, get into DEBUG, and type in the following program:

```
A>debug
-a1ØØ
Ø8F1:Ø1ØØ mov dx,5c
Ø8F1:Ø1Ø3 mov ah,f
Ø8F1:Ø1Ø5 int 21
Ø8F1:Ø1Ø7 int 2Ø
Ø8F1:Ø1Ø9
```

Here's what it looks like disassembled. (If you're using DOS version 1 you can use the "E" command to type in the hex codes for this program.) We've also added some comments about what each instruction does.

```
-u1ØØ,1Ø8
Ø8F1:Ø1ØØ BA5CØØ        MOV     DX,ØØ5C     ←— Put address of FCB in DX
Ø8F1:Ø1Ø3 B4ØF          MOV     AH,ØF       ←— Open File function
Ø8F1:Ø1Ø5 CD21          INT     21          ←— Call DOS
Ø8F1:Ø1Ø7 CD2Ø          INT     2Ø          ←— Return to DEBUG
```

In order to tell DOS what file we want to open, our program needs to put the filename in a place in memory agreed upon by both the program and the operating system. In sequential (and random) access this area of memory is called the *File Control Block*, or FCB.

> The File Control Block is a place in memory for passing messages about disk files between your program and the operating system.

The FCB is actually part of a larger area called the *Program Segment Prefix*. In a COM file, which is what DEBUG creates, the program starts at segment address 100h. The part of the segment below the program, from 0 to FFh, is the Program Segment Prefix. The FCB occupies the part of the Program Segment Prefix from 5Ch to 7Ch (in random access files the FCB goes up to 80h). This is shown in Figure 11-1.

Now that we have our little file-opening program in memory, let's take a look at the Program Segment Prefix so that we can see what happens to the FCB when we open a file. Before we look at it, however, we'll fill the entire FCB with a constant so we'll be able to tell what effect the Open File function has. We'll fill it with 11h, since we want to be able to tell if 00 has been written into it:

```
-f 4Ø 7f 11
```

We could have started our fill at 5Ch, but 40 will give us a broader picture. However, be careful not to start below 40h. Some locations below 40h in the Program Segment Prefix are used by DEBUG and the operating system (as we'll see in a moment), so erasing them will cause trouble.

Now we'll use D to see what the lower half of Program Segment Prefix looks like:

```
-dØ
Ø8F1:ØØØØ  CD 2Ø ØØ 2Ø ØØ 9A EE FE-1D FØ 42 Ø2 ØØ Ø6 7Ø Ø2
Ø8F1:ØØ1Ø  ØØ Ø6 E2 Ø4 34 Ø5 34 Ø5-Ø1 Ø1 Ø1 ØØ Ø2 FF FF FF
Ø8F1:ØØ2Ø  FF FF FF FF FF FF FF FF-FF FF FF FD Ø5 CA 2A
Ø8F1:ØØ3Ø  ØØ Ø6 ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ
Ø8F1:ØØ4Ø  11 11 11 11 11 11 11 11-11 11 11 11 11 11 11 11
Ø8F1:ØØ5Ø  11 11 11 11 11 11 11 11-11 11 11 11

                                                  11 11 11 11  } File
Ø8F1:ØØ6Ø  11 11 11 11 11 11 11 11-11 11 11 11 11 11 11 11  } Control
Ø8F1:ØØ7Ø  11 11 11 11 11 11 11 11-11 11 11 11 11 11 11 11  } Block
```

We've separated the FCB from the rest of the Program Segment Prefix to show you where it is. There are all the 11s we put in. Also, there are all sorts of mysterious numbers in the first few rows of the Program

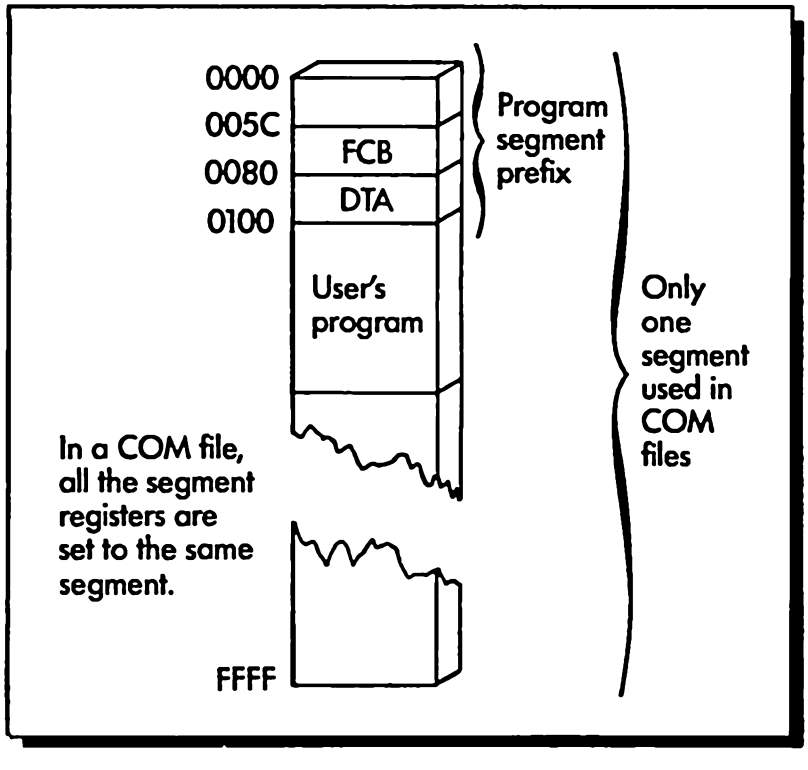
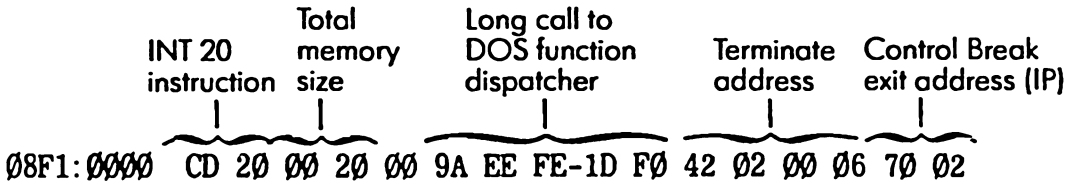


Figure 11-1. Program segment prefix in COM files

Segment Prefix. We don't really need to know what these numbers do, but we'll list their functions without further explanation just so you won't stay up all night wondering what these numbers are. Mostly they consist of addresses and information which the operating system (or DEBUG) needs to know about a particular file. It looks like this:



The structure of the Program Segment Prefix is shown in more detail in appendix E of the *IBM Personal Computer Disk Operating System* manual.

But to get back to opening a file, the question is: How do we put the name of the file which we want to open into the FCB? The answer is very simple. We use DEBUG's "N" command. Type "n", followed by the filename, with extension:

```
-ntestfile.txt
```

Now use "D" to see what's happened to the FCB:

```
-dØ
Ø9Ø5:ØØØØ  CD 2Ø ØØ 2Ø ØØ 9A FØ FF-ØD FØ 42 Ø2 ØØ Ø6 7Ø Ø2   M . ..p..pB...p.
Ø9Ø5:ØØ1Ø  ØØ Ø6 E2 Ø4 34 Ø5 ØØ Ø6-Ø1 Ø1 Ø1 ØØ Ø2 FF FF FF   ..b.4...........
Ø9Ø5:ØØ2Ø  FF FF FF FF FF FF FF FF-FF FF FF FF Ø2 Ø9 C8 2A   ..............H*
Ø9Ø5:ØØ3Ø  ØØ Ø6 ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø9Ø5:ØØ4Ø  11 11 11 11 11 11 11 11-11 11 11 11 11 11 11 11   ................
Ø9Ø5:ØØ5Ø  11 11 11 11 11 11 11 11-11 11 11 11

                                                ØØ 54 45 53   .............TES
Ø9Ø5:ØØ6Ø  54 46 49 4C 45 54 58 54-ØØ ØØ ØØ ØØ ØØ 2Ø 2Ø 2Ø   TFILETXT.....
Ø9Ø5:ØØ7Ø  2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø 2Ø-ØØ ØØ ØØ ØØ 11 11 11 11           ........

           Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

Well, would you look at that! The filename has been changed to all

capitals and filled into the FCB, starting at location 5D; and a bunch of 00s and 20s have been written there, starting with the 00 at location 5C and going up to 7B. What does it all mean?

Figure 11-2 shows all the locations in the FCB and what they do.

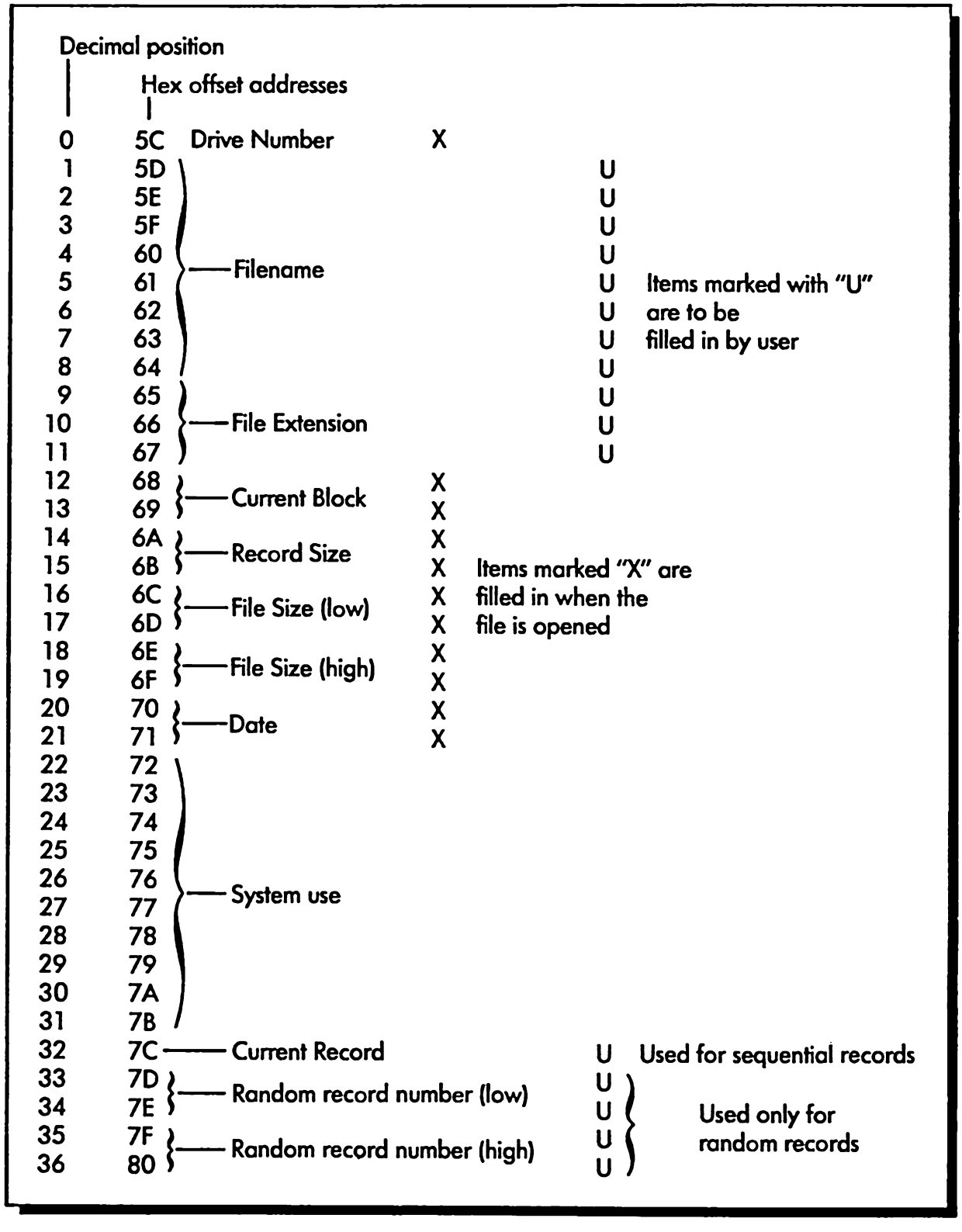


Figure 11-2. Arrangement of the file control block

We'll be explaining the various parts of this figure one by one as we go along. Stick a bookmark in the page — you'll be referring back to this figure frequently.

Now, finally, let's run our little DEBUG program, which will open the file TESTFILE.TXT. Running this program with the filename in the FCB, tells the operating system something like, "Track down TESTFILE.TXT and give me a rundown on it. I may want to be in touch with it later." To execute the program we simply use the "G" command:

```
-g
Program terminated normally
```

The red light on the disk drive will glow briefly, and you'll hear the disk drive whirr. The operating system has opened the file.

Now we can see what happened to the FCB:

```
-dØ
Ø9Ø5:ØØØØ  CD 2Ø ØØ 2Ø ØØ 9A FØ FF-ØD FØ 42 Ø2 ØØ Ø6 7Ø Ø2   M . ..p..pB...p.
Ø9Ø5:ØØ1Ø  ØØ Ø6 E2 Ø4 34 Ø5 ØØ Ø6-FF FF FF FF FF FF FF FF   ..b.4...........
Ø9Ø5:ØØ2Ø  FF FF FF FF FF FF FF FF-FF FF FF FF Ø2 Ø9 E6 FF   ..............f.
Ø9Ø5:ØØ3Ø  Ø5 Ø9 ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø9Ø5:ØØ4Ø  11 11 11 11 11 11 11 11-11 11 11 11 11 11 11 11   ................
Ø9Ø5:ØØ5Ø  11 11 11 11 11 11 11 11-11 11 11 11

                                                 Ø1 54 45 53   ............TES
Ø9Ø5:ØØ6Ø  54 46 49 4C 45 54 58 54-ØØ ØØ 8Ø ØØ 8Ø ØØ ØØ ØØ   TFILETXT........
Ø9Ø5:ØØ7Ø  F1 Ø6 1D 6E 4Ø 34 ØØ ØØ-ØØ 34 ØØ ØØ 11 11 11 11   q..n@4...4......

           Ø  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
```

Drive Number

What has the operating system found out for us about this file? First, the drive number at location 5C has been changed from 00 to 01. This means that the operating system has found the file on the current drive (the one in use at the moment), in this case drive A: (A = 1, B = 2, C = 3, and so on).

Records and Record Size

Second, there is now an 80h at location 6A, which is the location which specifies the record size. This is the default record size. We could have specified something else, but since we didn't, the operating system fills in 80h (128d). This brings up another topic: records.

Suppose you have a really big file on your disk, say the complete text

for a new novel. It just fits on the disk, but if you try to read it into memory it will be way too large. (Maybe the file is 300K long and you only have a 64K memory.) If you want to make some changes in the middle of the file, by reading it in and operating on it with your word processor, how can you do it? What's needed here is a way of dividing a file up into smaller parts, so we can work with manageable sections of it at a time, rather than trying to cram huge files into a small memory space. The answer, illustrated by Figure 11-3, is to use a *record*.

A record can be any convenient size. However, back when CP/M was being developed on 8-bit computers, an 80h (128d) byte record was chosen as a more or less standard size record. Now with 16-bit machines with much larger memories, this record size seems a little small, but it's still around as the default value for sequential files. ("Default" means that's the record size you'll get if you don't tell the operating system otherwise.) The record size number is stored in locations 6A and 6B of the FCB.

File Size

The *file size* at locations 6C through 6F has also been filled in, with 80h. This is the actual size in bytes of the entire file TESTFILE.TXT. (It turns out that 80h is the minimum file size generated by the word-processing program we used to create this file.) The file size consists of four bytes, and can therefore describe files up to about 4 billion bytes, which is large enough for almost anything. As you know, 16-bit numbers are stored in memory in reverse order: least significant bit first. Also, in double-word numbers, the order of the words is similarly reversed. Thus, because our file is 80h bytes long, the 80h appears as the *first* byte in the sequence instead of the last.

Date

The operating system has filled in locations 70h and 71h of the FCB

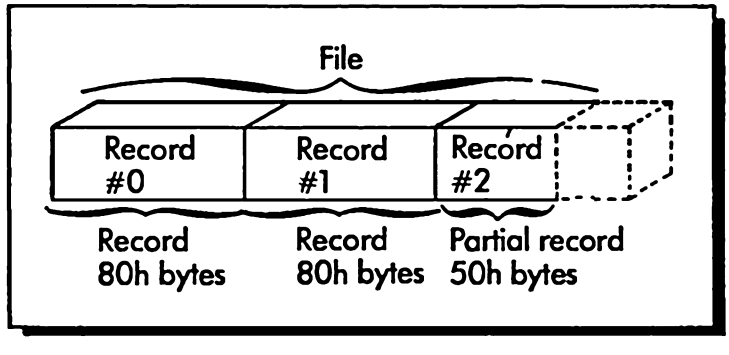
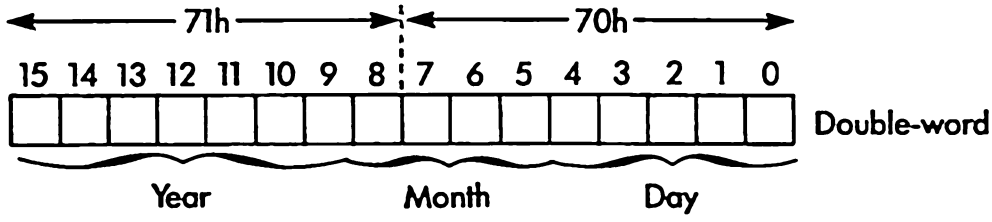


Figure 11-3. File divided into records

with the creation date of the file, expressed in a condensed format:

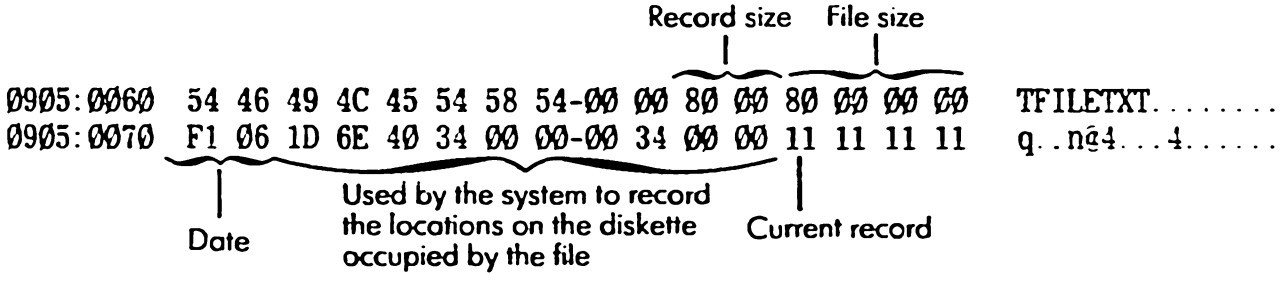


Possible values for the year run from 0 to 119d, representing 1980 to 2099. The month values run from 1 to 12d, and the day from 1 to 31d. Thus the F1 06 returned by the operating system tells us that the file was created on July 17, 1983. (Reversing the F1 and the 06 gives us 0000011-0111-10001 in binary, or 3-7-17 decimal, where the 3 means 1983.)

Mysterious File Location Bytes

The locations from 72h to 7Bh tell the operating system where the file is stored on the disk; however, the exact method for storing this information is complicated, and we don't need to get into it here.

In the figure below we've reproduced the last two lines of the DEBUG dump shown earlier, and labeled some of the important locations to summarize this discussion.



Reading a File

Now that we've opened a file, we can think about how to read its contents from the disk into the memory of the computer. We'll use DEBUG again to write a short program which not only opens a file, but also reads the first record of the file, using the Sequential Read DOS function.

SEQUENTIAL READ Function — Number 14h

Enter with:

Reg AH = 14h

Reg DS = segment address of open FCB

Reg DX = offset address of open FCB

The filename and extension, and the current block, current record, and record size must be entered in the FCB.

Execute:

```
INT 21
```

Return with:

Reg AL = 00 if record read successfully
= 01 if end-of-file, no data in record
= 02 if DTA too small, transfer ended
= 03 if end-of-file, partial record

Enter DEBUG and type in the following program:

```
A>debug
-a1ØØ
Ø8F1:Ø1ØØ mov dx,5c
Ø8F1:Ø1Ø3 mov ah,f
Ø8F1:Ø1Ø5 int 21
Ø8F1:Ø1Ø7 mov ah,14
Ø8F1:Ø1Ø9 int 21
Ø8F1:Ø1ØB int 2Ø
Ø8F1:Ø1ØD
```

If you want, you can save this program on your disk in the usual way.

```
-nreadrec.com
-rbx
BX ØØØØ
:
-rcx
CX ØØØØ
:d
-w
Writing ØØØD bytes
```

Unassemblying the program with "U" shows the following hex codes (we've added comments as well):

```
-u1ØØ,1Øc
Ø9Ø5:Ø1ØØ BA5CØØ        MOV     DX,ØØ5C     ←— Address of FCB into DX
Ø9Ø5:Ø1Ø3 B4ØF          MOV     AH,ØF       ←— Open File function
Ø9Ø5:Ø1Ø5 CD21          INT     21          ←— Call DOS
Ø9Ø5:Ø1Ø7 B414          MOV     AH,14       ←— Read Record function
Ø9Ø5:Ø1Ø9 CD21          INT     21          ←— Call DOS
Ø9Ø5:Ø1ØB CD2Ø          INT     2Ø          ←— Exit to DEBUG
```

Before you run this program you must deal with a few preliminaries. Put 11s in the FCB as before, and in the memory space from 80 to FF. Why? This brings up the topic of the Data Transfer Area.

The Data Transfer Area

As we mentioned earlier, there must be a place in memory to store the actual *data* to be transferred between your program and the disk. If your program is *reading* a file, the operating system reads this data off the disk and places it in memory. If you're *writing* to the disk, your program first places the data in memory and then calls DOS to write the data to the disk.

The part of memory used to transfer this data is called the "Data Transfer Area," or DTA.

> The Data Transfer Area is the place in memory where a record is stored on its way from or to a disk file.

With sequential and random files *only one record is transferred at a time.* Thus the DTA needs to be only as large as one record. We've already mentioned that the default size for a record is 80h (128d) bytes. As you might guess, the default size for the DTA is also 80h bytes. There is also a default *location* for the DTA. It starts at offset address 80h, and goes up to FFh. (Your program can change this location by using function 1Ah, "Set Disk Transfer Address," but we don't need to worry about that for the moment.)

Now that you know what the DTA is, you can fill both it and the FCB with 11s at the same time, in preparation for opening and reading a record.

```
-f 4Ø ff 11
```

The Current Record Number

The next thing we need to do is fill in the "current record number" in the FCB. If you refer to the parts of the FCB in Figure 11-2, you'll see that the current record number is at hex address 7Ch. What's the purpose of this number?

Since we've divided our file into records, and since we're going to read or write one record at a time into the DTA, we need to know *which record in the file* we're going to transfer. That's the purpose of the current record number. In sequential files we start at the first record in the file, which is number 0. So before we can read or write a file we need to put the number 0 into location 7Ch. We'll do that using the "E" command.

```
-e7c
Ø9Ø5:ØØ7C  11.Ø     <— type in 0 to set FCB to the beginning of the file
```

Finally we need to tell the operating system what file we want to access, so we use the "N" command as described above. (Remember that TESTFILE.TXT consists of a few short lines of prose created with your word processor or EDLIN.)

```
-ntestfile.txt
```

All set? Let's run the program!

```
-g
Program terminated normally
```

So far so good, but how do we know what happened? The "bottom line" of course is whether we can find the contents of the record in the DTA, since what we're trying to do is read a record. But first let's check the FCB to see what happened to *it*.

```
-dØ
Ø9Ø5:ØØØØ  CD 2Ø ØØ 2Ø ØØ 9A FØ FF-ØD FØ 42 Ø2 ØØ Ø6 7Ø Ø2   M . ..p..pB...p.
Ø9Ø5:ØØ1Ø  ØØ Ø6 E2 Ø4 34 Ø5 ØØ Ø6-FF FF FF FF FF FF FF FF   ..b.4...........
Ø9Ø5:ØØ2Ø  FF FF FF FF FF FF FF FF-FF FF FF FF Ø2 Ø9 E6 FF   ..............f.
Ø9Ø5:ØØ3Ø  Ø5 Ø9 ØØ ØØ ØØ ØØ ØØ ØØ-ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ   ................
Ø9Ø5:ØØ4Ø  11 11 11 11 11 11 11 11-11 11 11 11 11 11 11 11   ................
Ø9Ø5:ØØ5Ø  11 11 11 11 11 11 11 11-11 11 11 11

                                                  Ø1 54 45 53   ............TES
Ø9Ø5:ØØ6Ø  54 46 49 4C 45 54 58 54-ØØ ØØ 8Ø ØØ 8Ø ØØ ØØ ØØ   TFILETXT........
Ø9Ø5:ØØ7Ø  F1 Ø6 1D 6E 4Ø 34 ØØ ØØ-ØØ 34 ØØ ØØ Ø1 11 11 11   q..n@4...4......
                                                  |
                                                  Record number
```

Things are much the same as when we simply opened a file. However, there is one change: *the current record number has been incremented.* This is an important characteristic of sequential file access: each time a record is read, the current record number is automatically incremented to point to the next record in the file. Thus if you were reading a file with many records in it (rather than our little one-record demonstration), you could access the next record simply by repeating the Read Record function — there would be no need to change anything in the FCB.

So it's time for the moment of truth: is the record actually in the DTA? Let's dump it and see:

```
-d80
0905:0080  4E 6F 77 20 69 73 20 74-68 65 20 74 69 6D 65 0D  Now is the time.
0905:0090  0A 66 6F 72 20 61 6C 6C-20 67 6F 6F 64 20 6D 65  .for all good me
0905:00A0  6E 0D 0A 74 6F 20 63 6F-6D 65 20 74 6F 20 74 68  n..to come to th
0905:00B0  65 20 61 69 64 0D 0A 6F-66 20 74 68 65 69 72 20  e aid..of their
0905:00C0  63 6F 75 6E 74 72 79 2E-0D 0A 1A 1A 1A 1A 1A 1A  country.........
0905:00D0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A  ................
0905:00E0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A  ................
0905:00F0  1A 1A 1A 1A 1A 1A 1A 1A-1A 1A 1A 1A 1A 1A 1A 1A  ................
```

There it is! Just as we typed it in. Because we used a word processor to create this file, the entire 128d byte record is used, and the unwritten bytes are filled in with 1Ah, which is the end-of-file character. On the other hand, if you have used EDLIN to create this file, the record will be exactly as long as the prose you put into it; it will not be filled out with end-of-file characters, but will have only one.

If our file had been longer than 128d bytes it would have contained more than one record, and a second execution of the Read Sequential Record function would have been necessary to read the next record. However, it starts to get complicated to do this in DEBUG, so let's switch now to using the assembler, and see what a somewhat more sophisticated file-reading program looks like.

READFILE — Program to Read Text Files

The program shown below will read any text file and display it on the screen. It is very similar to the TYPE command built into the operating system, so it will give you some insight into how that command works.

For READFILE to function properly the file to be read must consist of text, and it must terminate with an end-of-file character, 1Ah, as the last character in the file. This tells the program to stop reading records in the file. EDLIN and most word processors automatically put one (or more) end-of-files at the end of a file, so this should not be a problem.

```
                                ;READFILE--Reads sequential records of file

= ØØ21                          doscall equ     21h     ;DOS interrupt number
= ØØØF                          openf   equ     Øfh     ;Open File function
= ØØ14                          readseq equ     14h     ;Read Sequential rec
= ØØØ2                          display equ      2h     ;Display Charac funct
= ØØ5C                          fcb     equ     5ch     ;file control block
= ØØ1A                          eof     equ     1ah     ;end-of-file character

                                ;*********************************************

ØØØØ                            datarea segment     ;program segment prefix

ØØ7C                                    org  7ch
ØØ7C  ??                        recno   db   ?          ;record number

ØØ8Ø                                    org  8Øh
ØØ8Ø     8Ø [                   dta     db   8Øh dup (?)  ;data transfer area
               ??
                   ]
Ø1ØØ                            datarea ends
                                ;*********************************************

ØØØØ                            pro_nam segment         ;define code segment

                                ;-----------------------------------------
ØØØØ                            main    proc    far     ;main part of program

                                        assume  cs:pro_nam,ds:datarea

ØØØØ                            start:          ;starting execution address

                                ;set up stack for return
ØØØØ  1E                               push    ds      ;save DS
ØØØ1  2B CØ                            sub     ax,ax   ;set AX to Ø
ØØØ3  5Ø                               push    ax      ;put it on stack

                                ;OPEN DISK FILE, SET RECORD NUMBER TO Ø
ØØØ4  BA ØØ5C                          mov  dx,fcb     ;set DX to FCB
ØØØ7  B4 ØF                            mov  ah,openf   ;Open File function
ØØØ9  CD 21                            int  doscall    ;call DOS
ØØØB  C6 Ø6 ØØ7C R ØØ                  mov  recno,Ø    ;put Ø in 7C

                                ;READ RECORD FROM FILE, sequential mode
ØØ1Ø                            read_rec:
ØØ1Ø  BA ØØ5C                          mov  dx,fcb     ;set DX to FCB
ØØ13  B4 14                            mov  ah,readseq ;Read Rec function
ØØ15  CD 21                            int  doscall    ;call DOS
```

```
                                ;PRINT RECORD FROM DISK TRANSFER AREA (DTA)
0017  B9 0080                          mov  cx,80h      ;number of chars in CX
001A  BB 0000                          mov  bx,0        ;initialize BX pointer
001D                            printit:
001D  8A 97 0080 R                     mov  dl,[dta + bx]  ;get character
0021  80 FA 1A                         cmp  dl,eof      ;end-of-file (1A) ?
0024  74 09                            je   exit        ;yes, so file finished

0026  B4 02                            mov  ah,display ;Display function
0028  CD 21                            int  doscall     ;call DOS
002A  43                               inc  bx          ;bump the pointer
002B  E2 F0                            loop printit     ;do 80h times
002D  EB E1                            jmp  read_rec    ;go get another record

002F  CB                        exit:  ret              ;return to DOS

0030                            main   endp     ;end of main part of program
                                ;-----------------------------------------
0030                            pro_nam ends    ;end of code segment
                                ;*****************************************

                                       end      start   ;end assembly
```

As you can see, this program uses the same DOS functions as the DEBUG version shown earlier: the Open File and Sequential Read functions. Each time the program reads a record it then prints out the contents on the screen, using the Display Character function. As it does this it checks each character to see if it's an end-of-file character (1Ah). When it finds one, the job is done, and it returns to DOS.

There are several interesting points to notice about this program. First, how does the filename — of the file we want to read — get into the FCB? Second, why don't we need to put the DATAREA data segment into the DS register? And finally, how can we get away with writing over part of the FCB with the DTA? Let's look at these questions in order.

How Does the Filename Get into the FCB?

In DEBUG we inserted (into the FCB) the filename of the file we wanted to read by using the "N" command, which was tailored for that very purpose. Now we have a stand-alone program: how does the filename get into the FCB?

The answer is simple: The operating system puts it there, *if we type it following the primary program name*. Thus if we wanted to read a file called NEWFILE.TXT, we'd call up READFILE like this:

```
A>readfile newfile.txt
```

This is the same way you tell utilities like TYPE, ERASE, and RENAME what file you want to operate on. As does the "N" command, the operating system not only fills the filename into the FCB, it also fills in the various other fields, such as the record number, with their initial values. This is a great convenience. Think how hard your program would have to work to get the filename from the keyboard, change it to all caps, and then write it into the FCB and set all the other fields to their initial values. DOS handles it all for you, out of the goodness of its heart.

Segments and the Program Segment Prefix

Did you notice in READFILE that we didn't put the data segment name, DATAREA, into the DS register as we've done in the past? Herein lies a subtle and complex tale.

When we used DEBUG to write our first file-reading program, the result was a COM file. The structure of a COM file is comparatively simple. All the segment registers are set to the same value. Thus there are no separate segments for data, extra segment, stack, and code. Everything is in the same segment. In a COM file, the *Program Segment Prefix* starts at location 0 in the segment, and extends up to location FF, for a total of 100h or 256d bytes. The FCB starts at 5Ch, and the DTA starts at 80h in the Program Segment Prefix. The program itself starts at location 100h. This was shown earlier in Figure 11-1.

In EXE files things are a bit more complicated. There can be separate segments for data, extra data, the stack, and the code. The question is: *Where is the Program Segment Prefix?* Is it in one of these segments? The answer is no. The Program Segment Prefix is in a location of its own. How do we find it? Fortunately, both the DS and ES registers are set to point to it when an EXE file is first loaded. The operating system (or DEBUG, if we're debugging the program) takes care of setting DS and ES to the right segment address. This arrangement is shown in Figure 11-4.

So for our program to access something in the Program Segment Prefix, it doesn't have to change the DS register at all. Of course, if we want to access data that our program has placed in the data segments, then we'll need to *change the DS register to point to this segment.* In some programs this involves switching the DS register back and forth repeatedly between its initial value (pointing to the Program Segment Prefix), and the value representing the address of the data segment in our program.

Interference Between the DTA and the FCB

You may have noticed that the FCB goes up to location 80h, and that

the DTA starts at location 80h, so there's an overlap of one byte, as shown in Figure 11-5.

Isn't this overlap a problem? Yes it is. The reasons for it are rooted in the development of CP/M, in those distant times when CP/M went from using sequential access to random access and there was no place else to put something called the "random record number." For sequential access there's no problem, because this number isn't used. However, for random access we have to be careful. Often the DTA has to be moved to a different location to avoid conflict. We'll say more about this later.

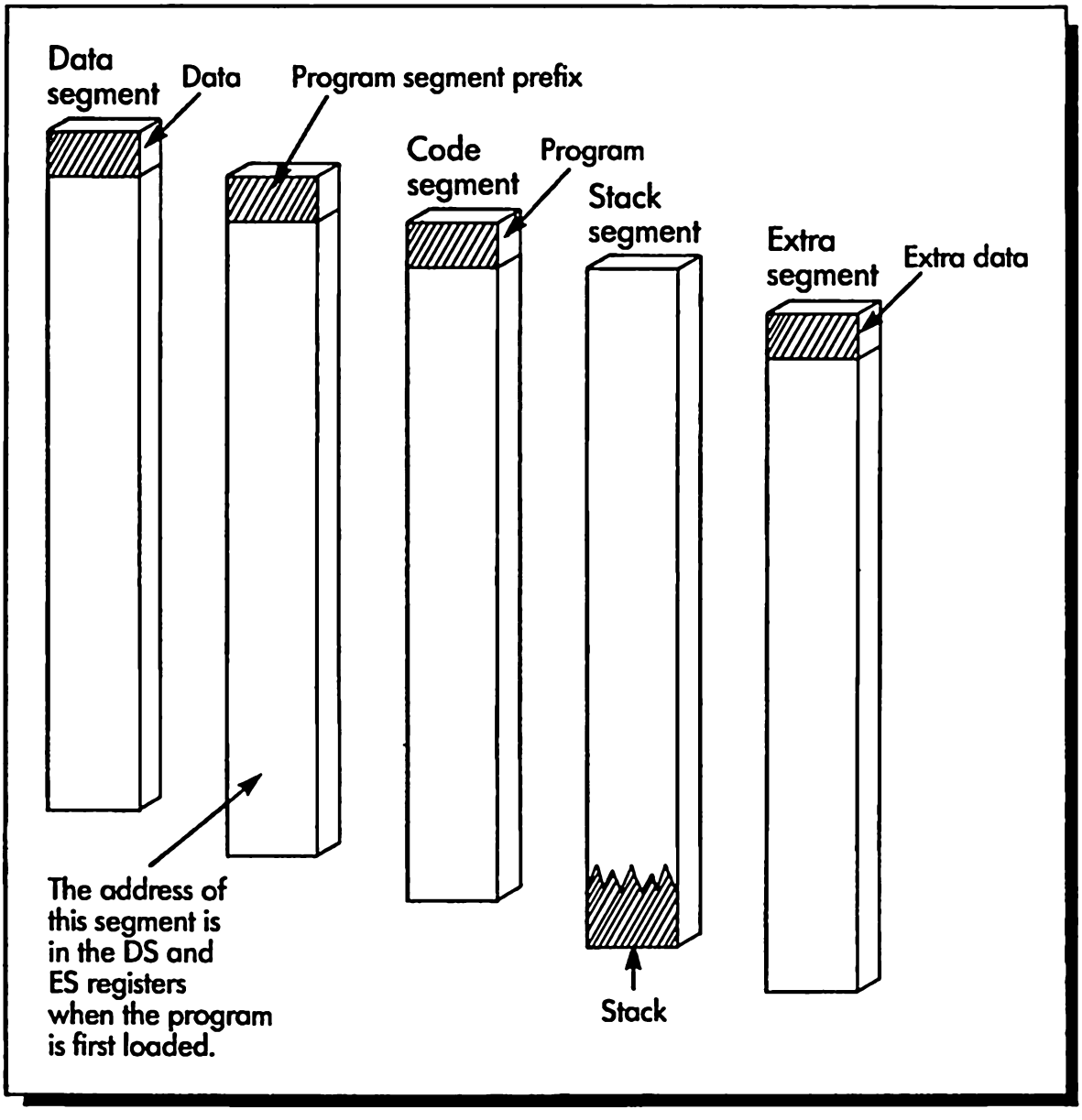


Figure 11-4. Program segment prefix in EXE files

Writing a Sequential File

Our next program is a simple one which takes sentences typed at the keyboard and writes them as records into a file on the diskette. Because it is such a simple program, we've taken certain shortcuts which make its operation a bit inelegant in some respects. We'll explain as we go along. Here's the program:

```
                                ;WRITE-F--Writes sequential records

= ØØ21                          doscall equ     21h     ;DOS interrupt number
= ØØ16                          create  equ     16h     ;Create File function
= ØØ15                          writesq equ     15h     ;Write Sequential rec
= ØØ1Ø                          close   equ     1Øh     ;Close File function
= ØØØA                          buffin  equ     Øah     ;buffered kbd input fn
= ØØ5C                          fcb     equ     5ch     ;file control block
= ØØØD                          return  equ     Ødh     ;ASCII carriage return
= ØØØA                          lfeed   equ     Øah     ;ASCII linefeed
                                ;*********************************************

ØØØØ                            datarea segment     ;program segment prefix

ØØ7C                                    org  7ch
ØØ7C  ??                        recno   db   ?              ;record number

ØØ8Ø                                    org  8Øh
ØØ8Ø     8Ø [                   dta     db   8Øh dup (?)  ;data transfer area
               ??
                   ]
Ø1ØØ                            datarea ends
                                ;*********************************************
```

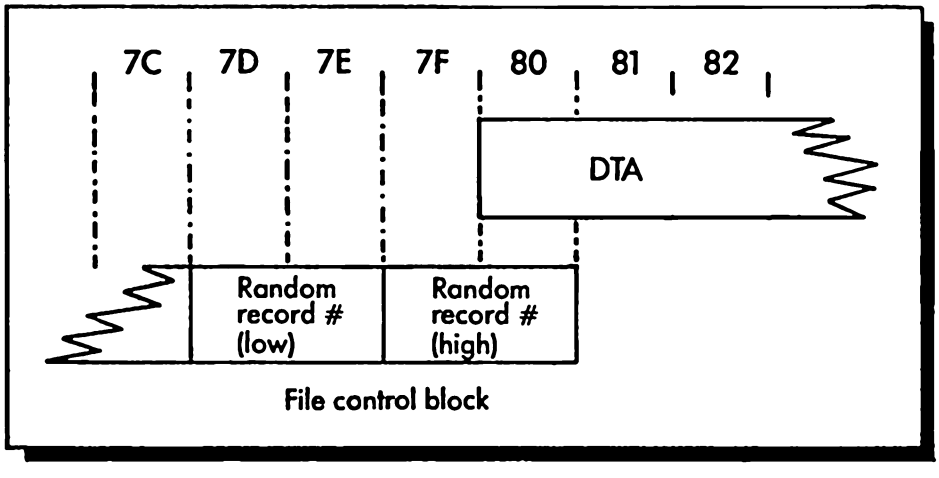


Figure 11-5. Overlap of FCB and DTA

```
0000                    pro_nam segment         ;define code segment

                        ;-----------------------------------------
0000                    main    proc    far     ;main part of program

                                assume  cs:pro_nam,ds:datarea

0000                    start:                  ;starting execution address

                        ;set up stack for return
0000  1E                        push    ds      ;save DS
0001  2B C0                     sub     ax,ax   ;set AX to 0
0003  50                        push    ax      ;put it on stack

                        ;CREATE DISK FILE, SET RECORD NUMBER TO 0
0004  BA 005C                  mov  dx,fcb     ;set DX to FCB
0007  B4 16                     mov  ah,create  ;Open File function
0009  CD 21                     int  doscall    ;call DOS
000B  C6 06 007C R 00           mov  recno,0    ;put 0 in 7C

                        ;BLANK BUFFER BY FILLING WITH RETURNS
0010                    newline:
0010  BB 0000                   mov  bx,0       ;first char in DTA
0013  B9 0080                   mov  cx,80h     ;CX is character count
0016                    erase:
0016  C6 87 0080 R 0D           mov  [bx + dta],return ;put cr in DTA
001B  43                        inc  bx         ;advance pointer
001C  E2 F8                     loop erase      ;repeat until done

                        ;GET LINE FROM KEYBOARD
001E  C6 06 007E R 4E           mov  dta-2,78   ;set max line length
0023  BA 007E R                 mov  dx,offset dta-2 ;addr of buffer
0026  B4 0A                     mov  ah,buffin  ;buffered keybd input
0028  CD 21                     int  doscall    ;call DOS
002A  80 3E 007F R 01           cmp  dta-1,1    ;if no chars typed,
002F  7E 14                     jle  exit       ;  then exit
                        ;insert linefeed following line of chars
0031  8A 1E 007F R              mov  bl,dta-1   ;put actual char count
0035  B7 00                     mov  bh,0       ;  into BX
0037  C6 87 0081 R 0A           mov  [dta+bx+1],lfeed ;insert linefeed

                        ;WRITE RECORD TO FILE, sequential mode
003C  BA 005C                   mov  dx,fcb     ;set DX to FCB
003F  B4 15                     mov  ah,writesq ;Sequent Write funct
0041  CD 21                     int  doscall    ;call DOS
0043  EB CB                     jmp  newline    ;go get another line
```

```
                                ;CLOSE FILE AND EXIT
0045                            exit:
0045  BA 005C                          mov  dx,fcb     ;put FCB address in DX
0048  B4 10                           mov  ah,close   ;Close File function
004A  CD 21                           int  doscall    ;call DOS
004C  CB                              ret             ;return to DOS

004D                            main    endp    ;end of main part of program
                                ;-----------------------------------------
004D                            pro_nam ends    ;end of code segment
                                ;*****************************************

                                        end     start   ;end assembly
```

WRITE-F is called at the same time as the filename to be written to.

```
A>write-f test1.txt
```

As with READFILE, this fills in the FCB with the filename, TEST1.TXT. When it's first started, the program sits there and waits for us to type a sentence. We type something, which must be less than one screen line (actually less than 78d characters), and then press ↵. Whenever we do this, the program takes the entire DTA and writes it into the file as a single record. Then it goes back and waits for us to do it again. If we don't type anything, but simply hit ↵ at the beginning of a line, the program knows we're done and returns to DOS.

As you can see, this program introduces several new function calls.

The Create File Function

When we wanted to read a file, we assumed the file already existed. To access an existing file, we *open* it. However, when we want to *write* to a file, we can assume that the file does not yet exist. (We might also want to write to an existing file: we'll cover that later.) To create a file which does not exist we use the *Create File* function.

CREATE FILE Function — Number 16h

Enter with:

Reg AH = 16h

Reg DS = segment address of FCB

Reg DX = offset address of FCB

The filename and extension must be entered in the File Control Block.

Execute:

```
INT 21
```

Return with:

Reg AL = 00 if creation is successful
= FFh if no space in directory

In the File Control Block, the drive number, current block, record size, file size and date are filled in.

If the file does not already exist, Create File creates it. If it *does* already exist, it is initialized to length zero, which destroys the previous contents of the file.

The Sequential Write Function

Writing to the diskette is similar to reading. Before we can write, the file must be opened or created, and the FCB has to be filled in accordingly. Each time we execute the Sequential Write function, a new record is written to the file and the record number is incremented. Actually the record may not be physically written to the disk every time this function is executed. If the record to be written isn't long enough to fill up an area of the diskette called a *sector*, it is placed in a buffer by the operating system. It stays there until there's a sector's worth of bytes, or until the file is closed; at which time all the accumulated records are written. You don't need to worry about this, since the operating system takes care of it automatically. A sector is 512d bytes, and is the primary unit of storage in the hardware-oriented world of the diskette drives.

SEQUENTIAL WRITE Function — Number 15h

Enter with:

Reg AH = 15h

Reg DS = segment address of open FCB

Reg DX = offset address of open FCB

The filename and extension, and the current block, current record, and record size must be entered in the FCB.
Execute:

```
INT 21
```

Return with:

Reg AL = 00 if record written successfully
= 01 if diskette is full
= 02 if DTA too small, transfer ended

The program is not as elegant as it might be. In what way? For one thing — in the interest of simplicity — there is no routine to print a carriage return and linefeed after each line is typed. As a consequence, each line you type overlays the previous one on the screen. We leave it to the reader to add a carriage return/linefeed routine. It could go after the section labeled "GET LINE FROM KEYBOARD."

Second, since the same length record is always written, and the length of the lines typed in can vary from 2 characters up to 78, the program fills out the balance of the record with carriage returns. These can't be seen on the screen, but they take time to print; and the delay is noticeable when a file which was created with this program is printed out using TYPE or READFILE. This is a harder problem to correct. One way would be to wait until the DTA was full before calling the Sequential Write function. This would require our program to do a certain amount of bookkeeping — to keep track of how full the DTA was, and so on — but would certainly be preferable in a serious program.

Try using this program to create a record. Call it up, with the name of the file you want to create. Then enter some lines of prose (each less than 78d characters).

```
A>write-f newfile.txt
Gather ye rosebuds while ye may,
Old Time is still a-flying:
And this same flower that smiles today
Tomorrow will be dying.
```

These lines will actually overprint the first one

Now use READFILE (or TYPE) to read the file back into memory and display it. It should contain all the lines you typed. Each of these lines is a separate record in the file, filled out with carriage returns to be 128d bytes long. (You won't notice the linefeeds, except for the delay in printing the lines.)

Record Lengths and the Operating System

Notice that once a file is stored on the disk, there is no way to tell what size records were used to create it. On the disk it's simply a file with a certain number of bytes. If you use DIR or any other program to look at the file, you won't find out anything about how many records are in the file or how long they are.

It's your *program* that decides what length records a particular file is to be divided into. It writes a certain number of records to the disk, which are then combined into a single file. Or it reads a certain number of records from a file, records whose length is again determined by the program. It's a little like ladling a cup of punch out of a big punch bowl. The cup holds one record's worth of punch. All the punch in the bowl is a file. You can use any size cup you want to take the punch out of the bowl, or to put it in, but once the punch is in the bowl it isn't divided into cups, it's just a big undivided mass.

You could write a file using records of one length, and read it back using records of another length (though why you would want to do this is not clear). The point is that your program can think of a file as being divided into records of any length it wants, as shown in Figure 11-6.

The Birthday Programs

In appendix B we've included three example programs which make use of sequential files to create and use a file of birth dates. These programs show how sequential files can be used to store formatted data. The techniques used in the birthday programs can be used to store other — perhaps more useful — kinds of similar information, such as names and addresses, sales figures, stock prices, and so forth.

The program SET-BD in appendix B lets you create a file of names and corresponding birthdays. You call up this program, then type in a name, a month, and a day; then another name, month and day; and so

forth until you're done. Each name/month/day group is an individual record. For efficiency, each record is 24d bytes long — that's smaller than the default value. This is an example of tailoring records for individual applications. (More information on operating the program is included in appendix B.)

Once you have the birthdays in a file, you can put them to use with the second program, called GET-BD. Every day when you first sit down at the computer you can type:

```
A>get-bd main.lst
```

(where MAIN.LST is the name of the file the birthdays are stored in). GET-BD will read the current date from the system clock, and then go through the file to see if anyone has a birthday with the current day's date. If it finds any, it prints out the names. You can then call those people to wish them happy birthday.

A third program, MOD-BD, is provided to modify the birthday file if, for instance, you find you've made a mistake, or you want to add some more names.

If you're not going to be using sequential file access, then you can skip these programs and forge ahead with the remainder of this chapter.

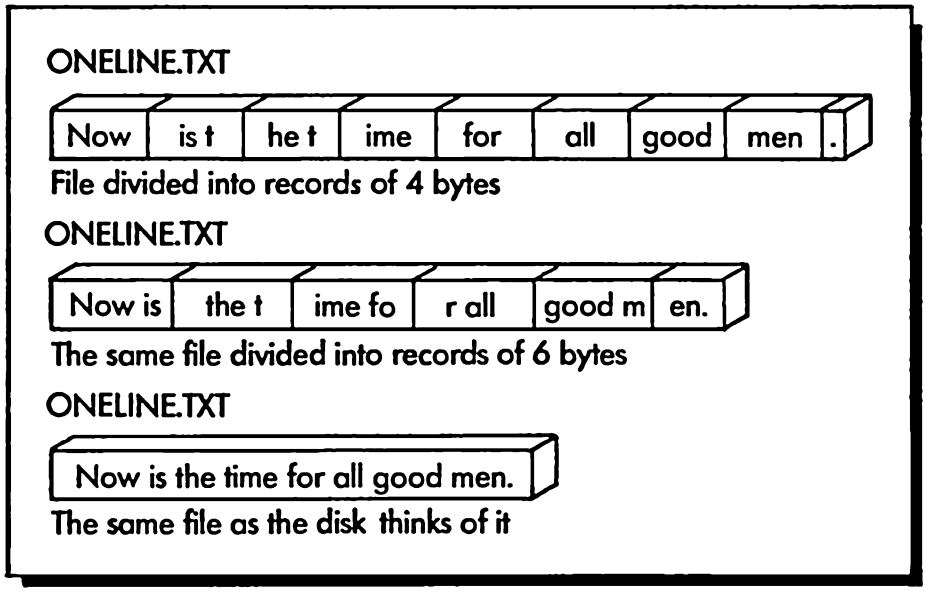


Figure 11-6. File divided into records in different ways

Random Access

The second way to read and write disk files is with the *random access* method. This is similar in many ways to sequential access, except that any record in a file may be accessed *directly* — there is no need to start at the beginning of the file and read all the records sequentially until you come to the one you want. The difference is shown in Figure 11-7.

The Random Record Number

One reason random records are easier to access is the way the random record number is structured. In sequential access there are actually *two* numbers used to specify the particular record in a file: the current record number at location 7Ch, and the current block number at locations 68h and 69h. When 127d records have been written to a file, the current record number is full. At this point a new "block" must be started, the current record number reset to zero, and the current block number incremented. This is somewhat awkward for your program to do (although it is possible).

The random record number, on the other hand, is simply a four-byte number. Thus random records start at 0 and go all the way up to more than 4 billion. (There isn't much chance you'll run out of record numbers.) Since the four bytes of the random record number constitute a double-word, they can be dealt with directly by assembly-language instructions, with no translation into blocks required.

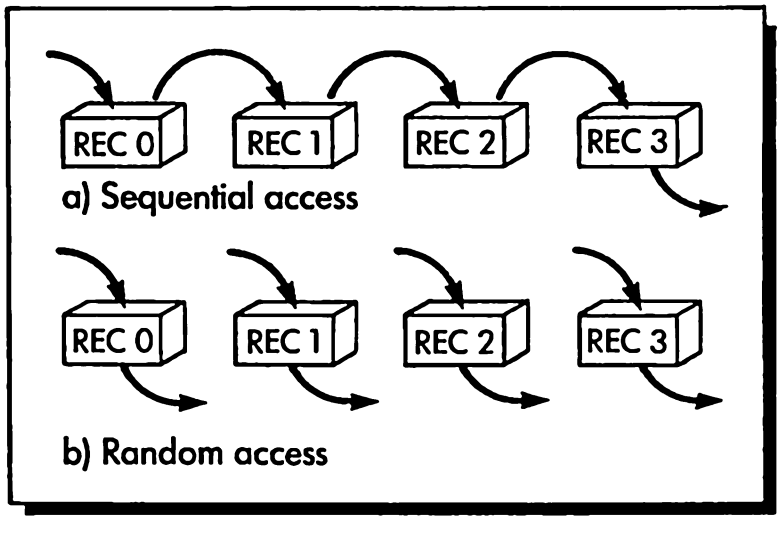


Figure 11-7. Random access and sequential access

The Random Read Function

RANDOM READ Function — Number 21h

Enter with:

Reg AH = 21h
Reg DS = segment address of open FCB
Reg DX = offset address of open FCB

The filename and extension, and the record size, and random record numbers must be entered in the FCB.

Execute:

```
INT 21
```

Return with:

Reg AL = 00 if record read successfully
= 01 if end-of-file, no data in record
= 02 if DTA too small, transfer ended
= 03 if end-of-file, partial record

The Random Read function is quite similar to the Sequential Read. However, one difference to keep in mind is that the random record number is *not* automatically incremented when a record is accessed, the way the current record number is. Thus, if you use the Random Read function to read a record, and then use it again without changing the random record number, you'll read exactly the same record again. In other words, *it is your program's responsibility to insert the correct random record number in the FCB before every random disk access (read or write).*

The READRAND Program

The program shown below reads a single random record from anywhere in the file. As with the other programs in this chapter, the file to be read from is specified by typing its name following "readrand." The record *number* to be read is specified by typing it in after the program is loaded. Here's the program:

```
                                ;READRAND--Reads random record in file
                                ;            record number is typed in by user

= 0021                          doscall equ     21h     ;DOS interrupt number
= 000F                          openf   equ     0fh     ;Open File function
= 0021                          readran equ     21h     ;Read Random Record
= 0002                          display equ      2h     ;Display Character fun
= 0001                          key_in  equ      1h     ;Keyboard Input funct
= 005C                          fcb     equ     5ch     ;file control block
= 001A                          eof     equ     1ah     ;end-of-file character

                                ;*********************************************

0000                            datarea segment         ;program segment prefix

007D                                    org  7dh        ;random rec numbers in FCB
007D  ????                      randlow dw   ?          ;random record number (lo)
007F  ????                      randhi  dw   ?          ;random record number (hi)

0080                                    org  80h
0080    80 [                    dta     db   80h dup (?)  ;data transfer area
            ??
               ]
0100                            datarea ends
                                ;*********************************************

0000                            pro_nam segment         ;define code segment

                                ;---------------------------------------------
0000                            main    proc    far     ;main part of program

                                        assume  cs:pro_nam,ds:datarea

0000                            start:                  ;starting execution address

                                ;set up stack for return
0000  1E                                push    ds      ;save DS
0001  2B C0                             sub     ax,ax   ;set AX to 0
0003  50                                push    ax      ;put it on stack

                                ;OPEN DISK FILE
0004  BA 005C                           mov     dx,fcb  ;set DX to FCB
0007  B4 0F                             mov     ah,openf ;Open File function
0009  CD 21                             int     doscall ;call DOS
```

```
                                ;GET RECORD NUMBER FROM KBD, PUT IN FCB
ØØØB  B4 Ø1                            mov  ah,key_in  ;Keyboard Input funct
ØØØD  CD 21                            int  doscall    ;call DOS
ØØØF  2C 3Ø                            sub  al,3Øh     ;convert to binary #
ØØ11  98                               cbw             ;convert byte to word
ØØ12  A3 ØØ7D R                        mov  randlow,ax ;put in FCB random low
ØØ15  C7 Ø6 ØØ7F R ØØØØ                mov  randhi,Ø   ;put Ø in random high

                                ;READ RECORD FROM FILE, random mode
ØØ1B  BA ØØ5C                          mov  dx,fcb     ;set DX to FCB
ØØ1E  B4 21                            mov  ah,readran ;Read Rec function
ØØ2Ø  CD 21                            int  doscall    ;call DOS

                                ;PRINT RECORD FROM DISK TRANSFER AREA (DTA)
ØØ22  B9 ØØ8Ø                          mov  cx,8Øh     ;number of chars in CX
ØØ25  BB ØØØØ                          mov  bx,Ø       ;initialize BX pointer
ØØ28                            printit:
ØØ28  8A 97 ØØ8Ø R                     mov  dl,[dta + bx]  ;get character
ØØ2C  8Ø FA 1A                         cmp  dl,eof     ;end-of-file (1A) ?
ØØ2F  74 Ø7                            je   exit       ;yes, so file finished
ØØ31  B4 Ø2                            mov  ah,display ;display function
ØØ33  CD 21                            int  doscall    ;call DOS
ØØ35  43                               inc  bx         ;bump the pointer
ØØ36  E2 FØ                            loop printit    ;do SØh times

ØØ38  CB                       exit:   ret             ;return to DOS

ØØ39                           main    endp    ;end of main part of program
                                ;-----------------------------------------
ØØ39                           pro_nam ends    ;end of code segment
                                ;*****************************************

                                        end     start   ;end assembly
```

To keep this program simple we've left out a routine to translate decimal numbers typed at the keyboard to binary in the program. Instead, we read a single character from the keyboard, which we assume is a digit from 0 to 9, and convert it to binary by subtracting 30h. Thus, record numbers higher than 9 cannot be accessed.

Also, to simplify the program we've left the DTA at its default address. Since we're using the random record number, which occupies locations from 7D to 80, there can be a conflict if we at the same time try to use location 80 as the first byte of the DTA. However, things work out all right in our program because we always zero out (write a zero to) the high part of the random record number *after* filling the DTA and *before* reading a record.

To demonstrate how this program works, we created a short prose

file, using EDLIN. Here's what it looks like, read back again using READFILE:

```
A>readfile duchess.txt
        "It hardly matters," said the Duchess, removing an
imaginary crumb from one of the antimacassars.  "After all, the
poor boy did have the temerity to question the established
order."
        "But surely," the vicar's face grew ashen, "that's not
reason enough to. . .I mean, the lions?"
        "Hush," the Duchess told him soothingly.  "The servants
will hear you."
```

Stored on the disk, this file — as we discussed before — is simply a string of characters. However, we'll assume that we want to access it in records of 80h bytes. READRAND uses this value for the record size, since it's the default value. Thus, each time we run the program, we'll see one 80h-byte record. Which record we see is determined by the digit we type after the program is loaded, as shown in the example session below.

```
A>readrand duchess.txt
1
have the temerity to question the established order."
        "But surely," the vicar's face grew ashen, "that's not
reason enough


A>readrand duchess.txt
Ø
       "It hardly matters," said the Duchess, removing an
imaginary crumb from one of the antimacassars.  "After all, the
poor boy did


A>readrand duchess.txt
2
 to. . .I mean, the lions?"
        "Hush," the Duchess told him soothingly.  "The servants
will hear you."
```

We asked for the second record of the file (number 1) first, then the first (number 0) and finally the last (number 2).

Writing Random Records

Writing random files is very similar to reading them, so we won't show an example program that does this. There are, however, several

possible pitfalls involved in using such a program. First, avoid writing noncontiguous records. What do we mean by that? Say you've written records 0, 1, 2, and 3. If you now write a record number 7, the operating system may become confused about the status of records 4, 5, and 6, and trouble can result.

Similarly, don't try to read nonexistent records. If you've only written 0, 1, and 2, you'll get unpredictable results if you try to read record 3.

Random Block Access

Random Block access makes it possible to read or write an entire file with one DOS function, instead of a single record. Random block access has the advantage of being simple to use: only one call to the function is necessary to read or write the file. On the other hand, it has the disadvantage that the Data Transfer Area which the file is read into (or written from) must be large enough to hold the entire file.

Of course, if the files you want to read are fairly small, and your system has enough memory, then this isn't a problem. And it's also possible to break a large file down into several blocks (each containing a number of records), and access one block at a time until the entire file has been transferred. In short, Random Block access provides a considerably more flexible means of file access than the more traditional sequential and random access methods. Random Block access also works with DOS versions 1 and 2. The only potential compatibility disadvantage is that should you ever want to translate your program to run under the CP/M operating system in use on many 8-bit computers, you would need to modify your file-accessing routines, since CP/M does not use Random Block access.

The Random Block Read Function

We're going to show only the Random Block Read function. Random Block Write is very similar, and can be easily figured out from the techniques used for Read.

RANDOM BLOCK READ Function — Number 27h

Enter with:

Reg AH = 27h

Reg DS = segment address of open FCB

Reg DX = offset address of open FCB

Reg CX = number of records to read

The filename and extension, and the record size, and random record numbers must be entered in the FCB.

Execute:

```
INT 21
```

Return with:

Reg AL = 00 if all records read successfully
= 01 if end-of-file, last record complete
= 02 if DTA too small, transfer ended
= 03 if end-of-file, last record partial

Reg CX = actual number of records read

As you can see, there is one important additional step to using this function: the number of records to be read must be placed in the CX register before the function is called. This is fine if we know in advance how big the file is, but what if we want to read an entire file of unknown size? One way would be to simply use a very large DTA and assume that the file would be small enough to fit. This would give an error return of AL=01. We could then figure out the size of the file by looking at CX, which returns with the actual number of records read.

The File Size Function

Another way to read a file of unknown size is to figure out in advance how large it is by using the File Size DOS function.

By using the File Size function we can set up our Random Block Read function to read exactly the right number of records at one time. We use this approach in the READBLOK program shown below. This program does just what READFILE and READRAND did; that is, it

reads a text file from the disk and displays it on the screen. In READBLOK, however, we can read the file with a single function call, so no looping is required in the program (except to print out the contents of the DTA on the screen).

FILE SIZE Function — Number 23h

Enter with:

Reg AH = 23h
Reg DS = segment address of unopened FCB
Reg DX = offset address of unopened FCB

The filename and extension, and record size must be entered in the FCB.

Execute:

```
INT 21
```

Return with:

Reg AL = 00 if file is found
= FFh if file not found

Random record file is set to *number of records in the file*, in terms of the record size specified on entry.

Note that we must set the record size field before calling File Size, and that we can't open the file until we've obtained its size. Before using Random Block Read we set the random record number (four bytes) to zero, which is the start of the file.

If we were going to divide a file into several blocks, then the random record field would be set automatically to point to the next record to be read, so it would not have to be changed for subsequent iterations of the Random Block Read call.

The READBLOK Program

The listing for READBLOK is shown below.

```
                                ;READBLOK--Read File, using Random Block Read
                                ;           Display file on screen

= ØØ21                          doscall equ     21h     ;DOS interrupt number
= ØØØF                          openf   equ     Øfh     ;Open File function
= ØØ23                          get_fs  equ     23h     ;get file size funct
= ØØØ2                          display equ      2h     ;display character fun
= ØØØ9                          print_m equ      9h     ;print message funct
= ØØ27                          block_r equ     27h     ;Random Block Read fun
= ØØ24                          set_ran equ     24h     ;set random rec field

= ØØ8Ø                          r_size  equ     8Øh     ;record size
= ØØ5C                          fcb     equ     5ch     ;File Control Block
= ØØ1A                          eof     equ     1ah     ;end-of-file character

                                ;*********************************************
ØØØØ                            datarea segment         ;define data segment

ØØ6A                                    org     6ah
 ØØ6A  ????                      rs_field dw     ?       ;rec size field in FCB


ØØ7D                                    org     7dh
ØØ7D  ????                      r1      dw      ?       ;random rec size (low)
ØØ7F  ????                      r2      dw      ?       ;random rec size (hi)

ØØ8Ø                                    org     8Øh     ;start of DTA
ØØ8Ø  4ØØØ [                   dta     db  4ØØØh dup (?) ;data transfer area
              ??
                 ]
4Ø8Ø  4E 6F 2Ø 73 75 63         mess1   db  'No such filename.$'
      68 2Ø 66 69 6C 65
      6E 61 6D 65 2E 24
4Ø92  42 61 64 2Ø 72 65         mess2   db  'Bad read.$'
      61 64 2E 24
4Ø9C                            datarea ends
                                ;*********************************************

ØØØØ                            pro_nam segment         ;define code segment

                                ;---------------------------------------------
ØØØØ                            main    proc    far     ;main part of program

                                        assume  cs:pro_nam,ds:datarea

ØØØØ                            start:                  ;starting execution address
```

```
                                ;set up stack for return
0000  1E                               push    ds      ;save DS
0001  2B C0                            sub     ax,ax   ;set AX to 0
0003  50                               push    ax      ;put it on stack

                                ;SET RECORD SIZE IN FCB
0004  C7 06 006A R 0080                mov  rs_field,r_size

                                ;GET FILE SIZE
000A  BA 005C                          mov  dx,fcb     ;FCB into DX
000D  B4 23                            mov  ah,get_fs  ;get file size funct
000F  CD 21                            int  doscall    ;call DOS
0011  FE C0                            inc  al         ;is AL=0? (was it FF?)
0013  74 3D                            jz   nofile     ;yes, so no file found
0015  8B 0E 007D R                     mov  cx,r1      ;put file size into CX

                                ;OPEN THE FILE
0019  BA 005C                          mov  dx,fcb     ;put FCB addr in DX
001C  B4 0F                            mov  ah, openf  ;Open File function
001E  CD 21                            int  doscall    ;call DOS

                                ;ZERO OUT RANDOM RECORD FIELD
0020  C7 06 007D R 0000                mov  r1,0       ;low byte
0026  C7 06 007F R 0000                mov  r2,0       ;high byte

                                ;READ BLOCK
                                ;(number of records still in CX)
002C  BA 005C                          mov  dx,fcb     ;put FCB address in DX
002F  B4 27                            mov  ah,block_r ;block read function
0031  CD 21                            int  doscall    ;call DOS
0033  0A C0                            or   al,al      ;check if read o.k.
0035  75 21                            jnz  bad_read   ;if AL not 0, bad read

                                ;PRINT OUT CONTENTS OF BUFFER
0037  B8 0080                          mov  ax,r_size  ;bytes/record in AX
003A  F7 E1                            mul  cx         ;# of records in CX
003C  8B C8                            mov  cx,ax      ;# of bytes now in CX
003E  BB 0000                          mov  bx,0       ;set pointer to 0
0041                            do_print:
0041  8A 97 0080 R                     mov  dl,[dta + bx]  ;get character
0045  80 FA 1A                         cmp  dl,eof     ;end-of-file (1A) ?
0048  74 07                            je   exit       ;yes
004A  B4 02                            mov  ah,display ;display function
004C  CD 21                            int  doscall    ;call DOS
004E  43                               inc  bx         ;bump pointer
004F  E2 F0                            loop do_print   ;do all characters
0051  CB                        exit:  ret             ;return to DOS
```

```
                                ;PRINT OUT MESSAGES
0052                            nofile:
0052  BA 40B0 R                         mov  dx, offset mess1  ;get message
0055  EB 07 90                          jmp  print_mess
0058                            bad_read:
0058  BA 4092 R                         mov  dx, offset mess2  ;get message
005B  EB 01 90                          jmp  print_mess
005E                            print_mess:
005E  B8  ---- R                        mov  ax,datarea ;put data segment
0061  8E D8                             mov  ds,ax      ;in DS register
0063  B4 09                             mov  ah,print_m ;print message funct
0065  CD 21                             int  doscall    ;call DOS
0067  CB                               ret             ;return to DOS

0068                            main    endp    ;end of main part of program
                                ;-----------------------------------------
0068                            pro_nam ends    ;end of code segment
                                ;*****************************************

                                        end     start   ;end assembly
```

In this program we've included some error messages. First, if the filename specified is not found, the "No such filename" message will be printed. You can try this out easily, by calling up the program with a nonexistent filename:

```
A>readblok nosuch.fn
No such filename.
A>
```

Also, if for any reason the read operation is less than successful, the "Bad Read" message will be printed.

Segment Management

There's an important detail to notice about the error-printing part of this program. Remember that when the operating system first loads the program it automatically sets the DS and ES registers to point to the Program Segment Prefix. They *don't* automatically point to the data segment. That's all right, as long as we don't want to access anything in the data segment. However, the error messages *are* in the data segment. How can we get at them?

No problem. We simply place the address of DATAREA (the data segment) in DS before trying to access our error messages. We do this in locations 005E and 0061.

Random Block Write and the SAVEIMAG Program

In appendix B you'll find a program called SAVEIMAG. This program takes the image of the monochrome screen, converts it to a file of ASCII characters, and writes it to the disk. It is analogous to the "screen dump" function which sends the screen image to the printer when ⇧ PrtSc are pressed, except that the image is saved as a file on the disk, rather than as a printout. This can be useful if, for example, you want to incorporate the screen image into a longer text file or manipulate it with your word processing program.

SAVEIMAG makes use of the Random Block Write function to write a buffer containing all the screen characters to the disk at one time. Since the number of characters is known in advance, we simply write enough records to include the full number of characters. The record size is the default value of 80h, and the number of characters to be written is 2050d, so the number of records (we let the assembler calculate it for us) is 10h. This is the number placed in CX before calling the Random Block Write function.

Summary

In this chapter you've learned about three methods of accessing files on the disk: sequential access, random access, and random block access. From the information in this chapter you should be able to solve almost any disk-access problem, with one exception: you can't deal with *pathnames*. Read on for a solution.

12

File Handle Disk Access

Concepts

File Handles
ASCIIZ strings
Access codes
File attributes

DOS Functions

(DOS version 2.00 only)
Open a File
Read from a File or Device
Create a File
Write to a File or Device
Close a File Handle
Move File Read/Write Pointer

In this chapter we're going to cover the last, and most sophisticated, of the four PC-DOS file access techniques: file handles. As we noted in the last chapter, the need for this new method arose because of the *pathnames* introduced in DOS 2.00. The file access techniques discussed in the last chapter all use the File Control Block to tell the operating system what filename to access, and the FCB has room only for an 8-character *filename* (plus 3-character extension), not a pathname. Thus a whole new system had to be created, and Microsoft, Inc. (the developers of PC-DOS) took advantage of this opportunity to develop a file access technique that was radically different from those that had gone before.

Features of File Handle Access

What's so different about file handle access? Three major features.

No More FCB

First, there is no more File Control Block. The various numbers in the FCB such as record size, file size, record number, and so forth, have been absorbed into DOS and are no longer "visible" to your program. To put it another way, you don't have to worry about them anymore. As for the filename itself, it has been replaced by the *pathname*, and the pathname has been made part of something called an ASCIIZ string. More on this later.

No More Records

The second major difference between file handle access and earlier systems is that the whole concept of *records* has disappeared. Now, only *bytes* are considered. A file is viewed as containing so many bytes, and if you want to read the file, that's how many bytes you read, using a single function call. Neither your program nor the operating system has to worry about records or record sizes. Random block access, discussed in the last chapter, went part way in this direction, but still operated in terms of records.

Introducing File Handles

The third way file handle access is different is that once a file has been opened, it is thenceforth referred to, not by its pathname — which can be long and cumbersome — but by a 16-bit *file handle*. Each open file is assigned a unique one. This file handle simply provides a shorthand way of referring to the file. Instead of all the Read, Write and other DOS functions having to communicate a long pathname to the operating system to access a file, they can use the file handle instead. Figure 12-1 shows — somewhat impressionistically — how this looks.

For those of you who are familiar with BASIC, there is some similarity between the file handle system used in PC-DOS, and BASIC's approach to disk files. In BASIC you first open a file with a statement like this:

```
100 OPEN "R", 2, "PROGNAM/EXT"
              |
          Buffer number
```

This statement opens a file called PROGNAM.EXT for random access, and assigns to the file a buffer number of 2. Subsequent access to this file is made using this buffer number, not the file name. Thus to read

something from record number R% in the file, the BASIC program would execute this statement:

```
200 GET 2, R%
        |
        Buffer number
```

Note how a single number has been assigned to the file, and is now used for reading and writing to the file. The *buffer number* in BASIC and the *file handle* in PC-DOS perform similar functions.

Other Features

There are other — less major — features of the file handle access technique. We'll mention them here briefly, then plunge right into an example program which opens a file, so we can see the ideas in action. Then we'll discuss all these aspects of file handle access in greater detail,

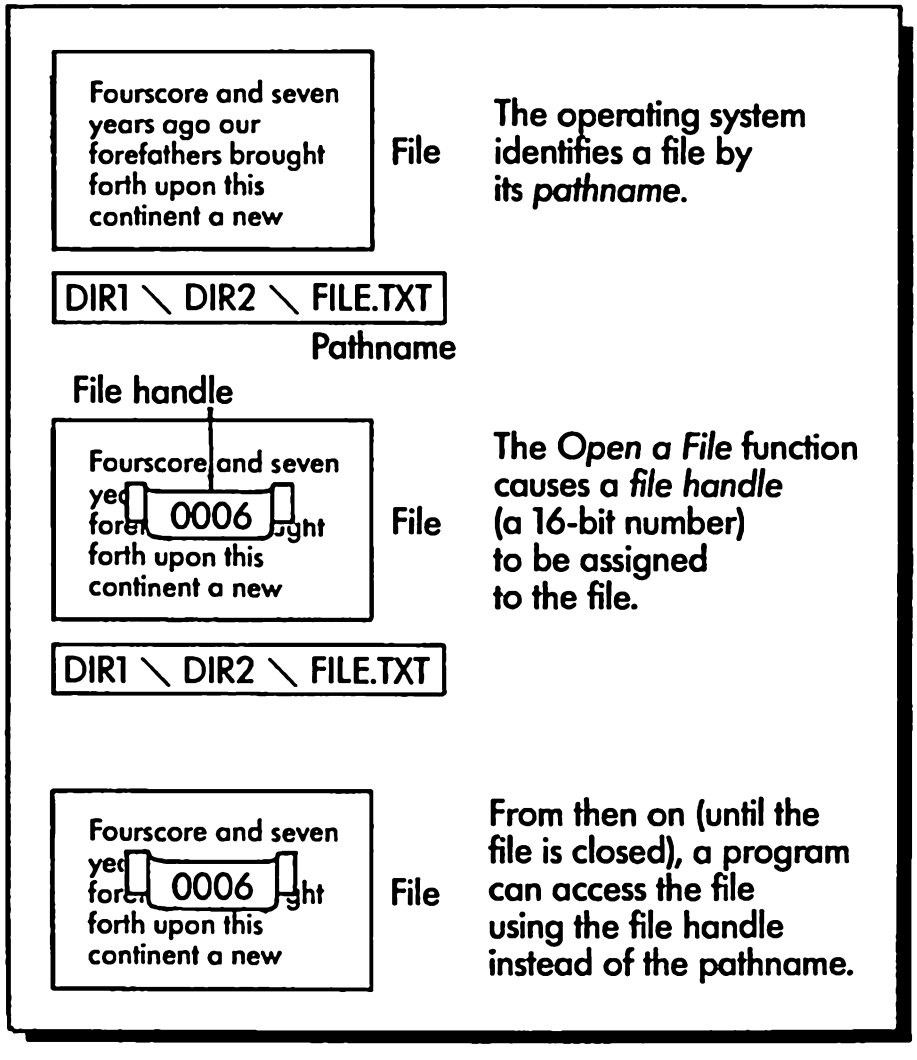


Figure 12-1. Pathnames and file handles

before moving on to programs that read and write files.

First, errors are handled in a more uniform way with file handle access than with the older disk access techniques. If an error of any kind occurs in *any* of the file handle DOS functions, the 8088 carry flag is set on return from the function. Thus a simple JC instruction is sufficient for your program to determine if an error has occurred. The AX register returns an error code, and the error codes are the same for all the functions, which makes it easier for your program to analyze them.

Second, when a file is opened it is given an *access code*. This determines whether the file is opened for reading, writing, or both. The access code can help to ensure that your program doesn't do something unexpected to the file, like write to a read only file.

Finally, when a file is first created it is given an *attribute*. This determines whether a file is read only, hidden, a system file, and so forth. We'll cover all this in more detail later.

The ZOPEN Program

The following program not only opens a file using the file handle technique, but also prints out a message and error code if there are any errors, and prints out the numerical value of the file handle if the file was opened successfully. By playing with this program we will begin to learn how to use the new file handle access technique; to get a handle on handles, so to speak.

To print out file handles and error codes in hex, the program incorporates the BINIHEX subroutine described in chapter 6.

Here's the program:

```
                                ;ZOPEN--Program to open file
                                ;          uses ASCIIZ format
                                ;          prints file "handle" if found
                                ;
                                ;*********************************************

0000                            datarea segment         ;define data segment

0000  31                        nambuff db  49          ;maximum bytes
0001  ??                                db  ?           ;bytes actually read
0002     32 [                           db  50 dup (?)  ;buffer
              ??
                 ]
0034  0D 0A 45 6E 74 65         intro   db  0dh,0ah,'Enter Pathname: $'
      72 20 50 61 74 68
```

```
      6E 61 6D 65 3A 2Ø
      24
ØØ47  45 72 72 6F 72 2Ø        emess   db  'Error $'
      24
ØØ4E  ØD ØA 24                  crlf    db  Ødh,Øah,'$' ;return and linefeed
ØØ51                            datarea ends
                                ;*****************************************

ØØØØ                            zopen   segment         ;define code segment

                                ;-----------------------------------------
ØØØØ                            main    proc    far     ;define main process

                                        assume  cs:zopen, ds:datarea

                                ;SET UP STACK FOR RETURN
ØØØØ                            start:
ØØØØ  1E                               push ds          ;save DS
ØØØ1  2B CØ                            sub  ax,ax       ;set AX to zero
ØØØ3  5Ø                               push ax          ;put it on stack


                                ;SET DS TO DATA BUFFER
ØØØ4  B8  ---- R                       mov  ax,datarea
ØØØ7  8E D8                            mov  ds,ax

                                ;READ IN PATHNAME OF FILE TO BE OPENED
ØØØ9                            newfile:
ØØØ9  BA ØØ34 R                        mov  dx, offset intro ;intro message
ØØØC  B4 Ø9                            mov  ah,9h       ;print message funct
ØØØE  CD 21                            int  21h         ;call DOS

ØØ1Ø  BA ØØØØ R                        mov  dx,offset nambuff ;addr of buffer
ØØ13  B4 ØA                            mov  ah,Øah      ;buff kbd input funct
ØØ15  CD 21                            int  21h         ;call DOS

ØØ17  BA ØØ4E R                        mov  dx,offset crlf ;return + linefeed
ØØ1A  B4 Ø9                            mov  ah,9h       ;print message functn
ØØ1C  CD 21                            int  21h         ;call DOS

                                ;INSERT ZERO IN BUFFER FOLLOWING NAME
ØØ1E  8A 1E ØØØ1 R                     mov  bl,nambuff+1 ;get # of bytes read
ØØ22  B7 ØØ                            mov  bh,Ø         ;  put in BX
ØØ24  C6 87 ØØØ2 R ØØ                  mov  [nambuff+bx+2],Ø ;zero into byte

                                ;SET DS:DX TO ASCIIZ STRING
ØØ29  BA ØØØ2 R                        mov  dx,offset nambuff+2 ;addr of name
```

```
                                ;SET AL TO ACCESS CODE
002C  B0 00                             mov  al,0       ;file open for reading

                                ;OPEN THE FILE
002E  B4 3D                             mov  ah,3dh     ;funct # to open file
0030  CD 21                             int  21h        ;call DOS

0032  8B D8                             mov  bx,ax      ;put "handle" in BX
0034  72 05                             jc   error      ;error return?
0036  E8 0046 R                        call binihex    ;no, print handle
0039  EB CE                             jmp  newfile    ;get another file

                                ;ERROR ROUTINE
003B                            error:
003B  BA 0047 R                         mov  dx, offset emess ;error message
003E  B4 09                             mov  ah,9h      ;funct # to print mess
0040  CD 21                             int  21h        ;call DOS
0042  E8 0046 R                        call binihex    ;print error number
0045  CB                                ret             ;return to DOS

0046                            main    endp            ;end main process
                                ;-----------------------------------------
                                ;
0046                            binihex proc    near
                                ;
                                ;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
                                ;  TO HEX ON CONSOLE SCREEN
                                ;
0046  B5 04                             mov     ch,4    ;number of digits
0048  B1 04                      rotate: mov     cl,4    ;set count to 4 bits
004A  D3 C3                             rol     bx,cl   ;left digit to right
004C  8A C3                             mov     al,bl   ;move to DL
004E  24 0F                             and     al,0fh  ;mask off left digit
0050  04 30                             add     al,30h  ;convert hex to ASCII
0052  3C 3A                             cmp     al,3ah  ;is it > 9 ?
0054  7C 02                             jl      printit ;no, so 0 to 9 digit
0056  04 07                             add     al,7h   ;yes, so A to F digit
0058                            printit:
0058  8A D0                             mov     dl,al   ;put ASCII char in DL
005A  B4 02                             mov     ah,2    ;display output funct
005C  CD 21                             int     21h     ;call DOS
005E  FE CD                             dec     ch      ;done 4 digits?
0060  75 E6                             jnz     rotate  ;not yet
0062  C3                                ret             ;done subroutine
                                ;
0063                            binihex endp
                                ;-----------------------------------------
0063                            zopen   ends            ;end code segment
                                ;*****************************************
                                        end  start      ;end assembly
```

If you want, you can type in this program and try running it now, even though we haven't yet explained how it works. All the program does is open the file whose pathname you type in following the program's prompt. If the file is opened without error, it prints the 16-bit file handle which DOS has assigned to the file; it then asks you for another pathname. If it can't find the file, or if some other error takes place, it prints "Error" followed by the error number, and returns to DOS. Let's try it out:

```
A>zopen

Enter Pathname: zopen.lst
0005
Enter Pathname: zread.lst
0006
Enter Pathname: zwrite.lst
0007
Enter Pathname: dir1\header.txt
0008
Enter Pathname: dir1\box.txt
0009
Enter Pathname: dir2\function.txt
Error 0004
```

The six pathnames we used all exist, so the program printed out the file handle assigned to each one, until the last. What happened there? The trouble is that only five files can be opened at once when using file handles. Error 4 is "too many open files, no handles left." (We'll talk more about errors later.)

If only five files can be opened at once it's not clear why 16-bit file handles are necessary: 8-bit handles would have been plenty large enough. However, 16-bit handles is what they are. Perhaps they leave room for future expansion.

What happens if we try to open a nonexistent file?

```
A>zopen

Enter Pathname: nosuch.fil
Error 0002
```

Error 2 is "file not found," which is just what you'd expect.

Now that we see *what* the program does, let's investigate in detail *how* it does it.

Pathnames

It's not quite as easy for our program to get hold of a pathname as it was to access the filename in the FCB. In ZOPEN, the program itself accepts the responsibility of getting the pathname from the user, first printing the "Enter Pathname:" prompt message.

The pathname could also have been obtained from the buffer at location 80h in the Program Segment Prefix. Everything we type in following the name of the file to be loaded is left in this buffer for access by our program. We can verify this using DEBUG:

```
A>debug zopen.exe \dir1\asmfiles\zopen.asm
-d80
09D5:0080  19 20 5C 64 69 72 31 5C-61 73 6D 66 69 6C 65 73   . \dir1\asmfiles
09D5:0090  5C 7A 6F 70 65 6E 2E 61-73 6D 0D 00 00 00 00 00   \zopen.asm......
09D5:00A0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
09D5:00B0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
09D5:00C0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
09D5:00D0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
09D5:00E0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
09D5:00F0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00   ................
```

There's the entire pathname in memory, waiting for our program to grab hold of it. However, for references to different pathnames from within the program, this is somewhat awkward because the Program Segment Prefix is in a different segment than the DATAREA data segment in which our program's messages are stored. To print a message we'd need to put the address of DATAREA in DS; but to read the pathname we'd have to put the value that was originally in DS back into DS. Switching back and forth like this is tedious, so it's more convenient to keep the pathname buffer in our own program's data segment.

The maximum pathname length allowed is 63d bytes, so we'll somewhat arbitrarily limit the buffer's size to 50d bytes. We'll use the Buffered Keyboard Input function to do the reading — this gives us full editing functions as we type the pathname.

ASCIIZ Strings

To open a file we must communicate the pathname of the file to the Open a File DOS function (number 3Dh). However, this function is actually not expecting the *pathname* as input. It is expecting a slight variation of the pathname called an *ASCIIZ string*. This is simply the pathname followed by a byte with a value of zero (hence the name ASCIIZ, for "ASCII-Zero"). The segment and offset addresses of the

resulting ASCIIZ string are then placed in the DS and DX registers, so that when the Open a File function is called it will know what pathname to open. See Figure 12-2.

The zero byte is inserted in the buffer following the pathname (see locations 001E to 0024 of the program) to form the ASCIIZ string. The DS register was set to DATAREA early in the program, so it points to the segment address of the ASCIIZ string; the offset address is set in location 0029.

Access Code

A file can be opened with one of three access codes:

0 — File is open for reading.

1 — File is open for writing.

2 — File is open for both reading and writing.

These codes tell the operating system things like, "I'm opening a file for reading only. If I try to write to it later, it's a mistake; don't let me do it."

When a file is created, it's given an *attribute*. We'll talk about attributes later, when we cover the *Create a File* function. However, one of the attributes that can be given a file is *read only*. If, in an attempt to open a file which has been given the read only attribute, your program uses an access code that permits writing, an error message 5, "Access denied," will be generated. Attributes and access codes together constitute a means of protecting files from being improperly read or written to.

The access code is set by the program using the instruction in location 002C. The code used is 0, which opens the file for reading.

The Open a File Function

Now that DS:DX is loaded with the address of the pathname and AL is set to the access code, we're ready to open the file. Note that the Open

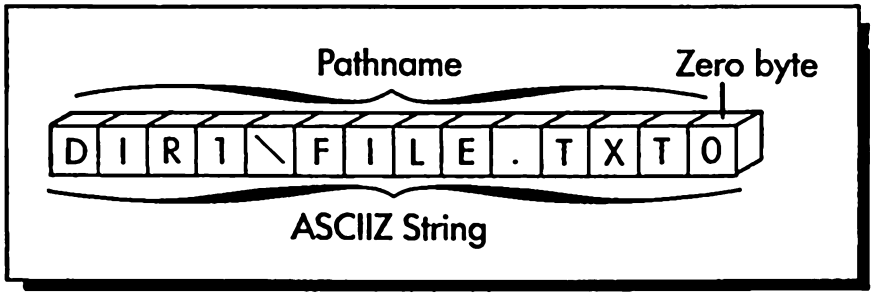


Figure 12-2. Pathname and ASCIIZ string

a File function, number 3Dh, is not the same as the earlier Open File function, number Fh, used for sequential and random access.

OPEN A FILE Function — Number 3Dh

Enter with:

Reg AH = 3Dh
Reg DS = segment address of ASCIIZ string
Reg DX = offset address of ASCIIZ string
Reg AL = access code

Execute:

```
INT 21
```

Return with:

Carry flag = 0 if operation successful
= 1 if error occurred
Reg AX = file handle if file opened successfully
= error code if error occurred

Possible errors are 2, 4, 5 and 12

When the Open a File function is executed, there are two possible results: either the file is found and opened successfully, or it isn't. If it is opened successfully, AX will contain the 16-bit file handle which will be used in all subsequent references to the file, such as reading it, writing to it, or closing it. Clearly your program should take care not to lose this number. It's like the claim check for your coat at a restaurant: if you lose it, you won't be able to access the file, or your coat, again.

File Handles

As we've seen, there are only so many file handles. Also, we've seen that the handle numbers seem to start with 0006, rather than with 0000 as you'd expect. Why? The handle numbers 0000 to 0005 are predefined by DOS, and already stand for various input and output devices, as shown in the list below:

Handle — Predefined Files

0000 — Standard input device. Input can be redirected.

0001 — Standard output device. Output can be redirected.

0002 — Standard error output device. Output cannot be redirected.

0004 — Standard auxiliary device.

0005 — Standard printer device.

These predefined file handles are used with *redirection*, the advanced PC-DOS feature that lets you take the input or output which would normally go one place, and redirect it someplace else, say to the printer instead of a disk file. We're not going to discuss redirection here; for an explanation of redirection and other DOS advanced features, see *DOS Primer for the IBM PC and XT* by Mitchell Waite, John Angermeyer, and Mark Noble (New York: Plume/Waite, New American Library, 1984).

If the carry flag is set on return from opening the file, it means an error has occurred.

Errors

Errors are handled much more uniformly in the file handle functions than in the sequential and random functions. In all the file handle functions, if the carry flag is set on completion of the call, it means that an error has occurred. A zero in the carry flag means the operation was successful. If an error occurred, the AX register always contains the error code, and these codes are all taken from the same list (although not all the errors on the list are possible with a given function).

Here's the list of error codes:

Decimal Code	Hex Code	Error Description
1	01	Invalid function number
2	02	File not found (a common error)
3	03	Path not found
4	04	Too many open files (no handles left)
5	05	Access denied (wrong attribute or access code)
6	06	Invalid handle
7	07	Memory control blocks destroyed
8	08	Insufficient memory
9	09	Invalid memory block address
10	0A	Invalid environment
11	0B	Invalid format
12	0C	Invalid access code

13	0D	Invalid data
14	0E	(not used)
15	0F	Invalid drive specified
16	10	Attempted to remove the current directory
17	11	Not same device
18	12	No more files

Many of these codes sound like damage reports from a space mission ("Memory control blocks destroyed, Captain"), but others, such as "File not found," are quite common. Since the same codes are used by all the functions, your program can use a single subroutine to figure out what error occurred and print out an appropriate message, rather than having separate error routines for Open a File, Read File, and so on.

By now you should have a grasp of file handles, pathnames, and ASCIIZ strings, and how they fit together. Let's see how we'd read a file, using these ideas.

The ZREAD Program

The program below does just what the various READ programs in the last chapter did, and what the TYPE function in DOS does: It reads a text file from the disk and displays the results on the screen. After loading, the program — like the ZOPEN program — prints "Enter Pathname" and waits for you to do just that. It then tries to open the file. If the file is opened successfully. the program reads its contents, using the *Read from a File or Device* function, and prints it out on the screen. Here's the program:

```
                                ;ZREAD--Program to read file
                                :          uses ASCIIZ format
                                :
                                ;*******************************************

0000                            datarea segment         ;define data segment

0000  31                        nambuff db  49          ;maximum bytes
0001  ??                                db  ?           ;bytes actually read
0002    32 [                            db  50 dup (?)  ;name buffer
           ??
               ]
0034    C8 [                    datbuff db  200 dup (?) ;data buffer
           ??
               ]
00FC  0D 0A 45 6E 74 65         intro   db  0dh,0ah,'Enter Pathname: $'
      72 20 50 61 74 68
```

```
      6E 61 6D 65 3A 2Ø
      24
Ø1ØF  45 72 72 6F 72 2Ø       emess   db  'Error $'
      24
Ø116  ØD ØA 24                crlf    db  Ødh,Øah,'$' ;return and linefeed
Ø119                          datarea ends
                              ;*****************************************

ØØØØ                          zread   segment         ;define code segment

                              ;-----------------------------------------
ØØØØ                          main    proc    far     ;define main process

                                      assume  cs:zread, ds:datarea

                              ;set up stack for return
ØØØØ                          start:
ØØØØ  1E                             push ds          ;save DS
ØØØ1  2B CØ                          sub  ax,ax       ;set AX to zero
ØØØ3  5Ø                             push ax          ;put it on stack

                              ;set DS to data segment
ØØØ4  B8  ---- R                     mov  ax,datarea
ØØØ7  8E D8                          mov  ds,ax

                              ;READ IN PATHNAME OF FILE TO BE OPENED
ØØØ9                          newfile:
ØØØ9  BA ØØFC R                      mov  dx, offset intro ;intro message
ØØØC  B4 Ø9                          mov  ah,9h       ;print message funct
ØØØE  CD 21                          int  21h         ;call DOS

ØØ1Ø  BA ØØØØ R                      mov  dx,offset nambuff ;addr of buffer
ØØ13  B4 ØA                          mov  ah,Øah      ;buff kbd input funct
ØØ15  CD 21                          int  21h         ;call DOS

ØØ17  BA Ø116 R                      mov  dx,offset crlf ;return + linefeed
ØØ1A  B4 Ø9                          mov  ah,9h       ;print message functn
ØØ1C  CD 21                          int  21h         ;call DOS

                              ;Insert zero in buffer following name
ØØ1E  8A 1E ØØØ1 R                   mov  bl,nambuff+1 ;get # of bytes read
ØØ22  B7 ØØ                          mov  bh,Ø         ;  put in BX
ØØ24  C6 87 ØØØ2 R ØØ                mov  [nambuff+bx+2],Ø ;zero into byte

                              ;OPEN FILE
ØØ29  BA ØØØ2 R                      mov  dx,offset nambuff+2 ;addr of name
ØØ2C  BØ ØØ                          mov  al,Ø        ;file open for reading
ØØ2E  B4 3D                          mov  ah,3dh      ;funct # to open file
```

```
ØØ3Ø  CD 21                    int  21h        ;call DOS
ØØ32  8B D8                    mov  bx,ax      ;handle/error in BX
ØØ34  72 2B                    jc   error      ;error return?

                        ;READ FILE
ØØ36                    newbuff:
ØØ36  B9 ØØC8                  mov  cx,2ØØ     ;# of bytes to read
ØØ39  BA ØØ34 R                mov  dx,offset datbuff ;addr of buffer
ØØ3C  B4 3F                    mov  ah,3fh     ;read from file functn
ØØ3E  CD 21                    int  21h        ;call DOS
ØØ4Ø  72 1F                    jc   error      ;error return?
ØØ42  3D ØØØØ                  cmp  ax,Ø       ;no, are we at EOF ?
ØØ45  74 19                    je   exit       ;yes, exit

                        ;DISPLAY BUFFER CONTENTS
ØØ47  8B F3                    mov  si,bx      ;save handle
ØØ49  8B C8                    mov  cx,ax      ;# chars read in CX
ØØ4B  BB ØØ34 R                mov  bx,offset datbuff ;addr of buffer
ØØ4E                    newchar:
ØØ4E  B4 Ø2                    mov  ah,2       ;display output funct
ØØ5Ø  8A 17                    mov  dl,[bx]    ;get character
ØØ52  8Ø FA 1A                 cmp  dl,1ah     ;is it end-of-file ?
ØØ55  74 Ø9                    je   exit       ;yes, exit
ØØ57  CD 21                    int  21h        ;call DOS, display it
ØØ59  43                       inc  bx         ;point to next char
ØØ5A  E2 F2                    loop newchar    ;done all chars?
ØØ5C  8B DE                    mov  bx,si      ;yes, restore handle
ØØ5E  EB D6                    jmp  newbuff    ;go fill buffer again
ØØ6Ø                    exit:
ØØ6Ø  CB                       ret             ;return to DOS

                        ;ERROR ROUTINE
ØØ61                    error:
ØØ61  BA Ø1ØF R                mov  dx, offset emess ;error message
ØØ64  B4 Ø9                    mov  ah,9h      ;funct # to print mess
ØØ66  CD 21                    int  21h        ;call DOS
ØØ68  E8 ØØ6C R                call binihex    ;print error number
ØØ6B  CB                       ret             ;return to DOS

ØØ6C                    main    endp            ;end main process
                        ;-----------------------------------------
                        ;
ØØ6C                    binihex proc    near
                        ;
                        ;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
                        ;  TO HEX ON CONSOLE SCREEN
                        ;
ØØ6C  B5 Ø4                    mov     ch,4    ;number of digits
ØØ6E  B1 Ø4             rotate: mov     cl,4    ;set count to 4 bits
```

```
ØØ7Ø  D3 C3                     rol     bx,cl   ;left digit to right
ØØ72  8A C3                     mov     al,bl   ;move to DL
ØØ74  24 ØF                     and     al,Øfh  ;mask off left digit
ØØ76  Ø4 3Ø                     add     al,3Øh  ;convert hex to ASCII
ØØ78  3C 3A                     cmp     al,3ah  ;is it > 9 ?
ØØ7A  7C Ø2                     jl      printit ;no, so Ø to 9 digit
ØØ7C  Ø4 Ø7                     add     al,7h   ;yes, so A to F digit
ØØ7E                    printit:
ØØ7E  8A DØ                     mov     dl,al   ;put ASCII char in DL
ØØ8Ø  B4 Ø2                     mov     ah,2    ;display output funct
ØØ82  CD 21                     int     21h     ;call DOS
ØØ84  FE CD                     dec     ch      ;done 4 digits?
ØØ86  75 E6                     jnz     rotate  ;not yet
ØØ88  C3                        ret             ;done subroutine
                        ;
ØØ89                    binihex endp
                        ;-----------------------------------------
ØØ89                    zread   ends            ;end code segment
                        ;*****************************************
                                end  start      ;end assembly
```

ZREAD first obtains the pathname and turns it into an ASCIIZ string to open the file just as ZOPEN did. Once the file is opened, the program reads it.

The Read from a File or Device Function

Here's where we use the file handle that was returned in the AX register when we opened the file. The *Read from a File or Device* function requires the file handle to be in BX, so we MOVe it over. The address of the buffer we're going to put the file in is placed in DS:DX. We also must tell the function how many bytes to read. This can be almost any number we like, as long as we have room for the buffer in memory. Telling it to read more bytes than are in the buffer is an invitation to have a text file written on top of your program.

There are all sorts of things your program can find out about what's happened on its return from the *Read from a File or Device* function. If an error occurred, AX will return error code 5 or 6. If the operation was successful, AX will return the number of bytes that were actually read into the buffer. If it returns zero, no bytes were read; there was nothing to read but the end-of-file.

Notice that this function must be repeated each time the number of bytes specified in CX is read into the buffer. In this respect it's more like the Sequential Read function than the Read Random Block function of the last chapter. Of course, if you already know how large the file is, and

you have room for it in memory, you can read it all in with a single call to this function.

Since the function will read an end-of-file character into the buffer as just another character, the program needs to check for this character and terminate the program if it finds it.

READ FROM A FILE OR DEVICE Function Number 3Fh

Enter with:

Reg AH = 3Fh
Reg BX = the file handle
Reg CX = number of bytes to read
Reg DS = segment address of data buffer
Reg DX = offset address of data buffer

Execute:

```
INT 21
```

Return with:

Carry flag = 0 if operation successful
= 1 if error occurred

Reg AX = number of bytes actually read
= 0 if only end-of-file found
= error code if error occurred

Possible errors are 5 and 6

Something else to notice is that we need to keep the file handle in BX, so that each time we call the *Read from a File or Device* function it will access the right pathname. Since the program also uses BX as a pointer when printing out the contents of the buffer, it's necessary to save BX somewhere while this is going on. We save it in the SI register. We could also have saved it by PUSHing it, but this leads to complications because of the multiple exits from the interior of the "newchar" loop. Remember, whatever is PUSHed must get POPped. It's bad form to exit from a program without POPping everything you PUSHed.

Now that we know how to read files using file handles, what about writing to them?

Writing to a File

Here's a program that uses file handles to let you type in a prose file from the keyboard. You call the program, and when it executes it says "Enter Pathname." You type in the name of the file you want your text written to. Then it says "Enter text." You can type in as many lines of text as you want. Each line must be less than 80 characters, and must be terminated with ↵. Typing ↵ at the beginning of a line terminates the program, as you can see from the sample session:

```
A>zwrite
Enter Pathname: dir1\file1.txt ————— Pathname entered by user
Enter text:
Once upon a midnight dreary,                                  }
While I pondered, weak and weary,                             } Text entered by user
Over many a quaint and curious volume of forgotten lore,      }
↵ typed by user to terminate program
```

Here's the program:

```
                                ;ZWRITE--Program to write file
                                ;            from keyboard input
                                ;          Uses ASCIIZ format
                                ;
                                ;*******************************************

ØØØØ                            datarea segment         ;define data segment

ØØØØ  32                        nambuff db  5Ø          ;max pathname bytes
ØØØ1  ??                                db  ?           ;bytes actually read
ØØØ2     32 [                           db  5Ø dup (?)  ;name buffer
               ??
                  ]
ØØ34  5Ø                        datbuff db  8Ø          ;maximum text bytes
ØØ35  ??                                db  ?           ;bytes actually typed
ØØ36     5Ø [                           db  8Ø dup (?)  ;text buffer
               ??
                  ]
ØØ86  ????                      handle  dw  ?           ;handle storage
ØØ88  ØD ØA 45 6E 74 65         intro   db  Ødh,Øah,'Enter Pathname: $'
      72 2Ø 5Ø 61 74 68
      6E 61 6D 65 3A 2Ø
      24
ØØ9B  ØD ØA 45 6E 74 65         intro2  db  Ødh,Øah,'Enter Text:',Ødh,Øah,'$'
      72 2Ø 54 65 78 74
      3A ØD ØA 24
```

```
00AB  45 72 72 6F 72 20        emess   db  'Error $'
      24
00B2  0D 0A 24                 crlf    db  0dh,0ah,'$' ;return and linefeed
00B5                           datarea ends
                               ;*****************************************

0000                           zread   segment         ;define code segment

                               ;-----------------------------------------
0000                           main    proc    far     ;define main process

                                       assume  cs:zread, ds:datarea

                               ;set up stack for return
0000                           start:
0000  1E                               push ds         ;save DS
0001  2B C0                            sub  ax,ax      ;set AX to zero
0003  50                               push ax         ;put it on stack

                               ;set DS to data segment
0004  B8  ---- R                       mov  ax,datarea
0007  8E D8                            mov  ds,ax

                               ;READ IN PATHNAME OF FILE TO BE OPENED

0009  BA 0088 R                        mov  dx, offset intro ;intro message
000C  B4 09                            mov  ah,9h      ;print message funct
000E  CD 21                            int  21h        ;call DOS

0010  BA 0000 R                        mov  dx,offset nambuff ;addr of buffer
0013  B4 0A                            mov  ah,0ah     ;buff kbd input funct
0015  CD 21                            int  21h        ;call DOS

0017  BA 00B2 R                        mov  dx,offset crlf ;return + linefeed
001A  B4 09                            mov  ah,9h      ;print message functn
001C  CD 21                            int  21h        ;call DOS

                               ;Insert zero in buffer following name
001E  8A 1E 0001 R                     mov  bl,nambuff+1 ;get # of bytes read
0022  B7 00                            mov  bh,0       ;  put in BX
0024  C6 87 0002 R 00                  mov  [nambuff+bx+2],0 ;zero into byte

                               ;CREATE FILE
0029  BA 0002 R                        mov  dx,offset nambuff+2 ;addr of name
002C  B9 0000                          mov  cx,0       ;normal attribute
002F  B4 3C                            mov  ah,3ch     ;create file function
0031  CD 21                            int  21h        ;call DOS
0033  A3 0086 R                        mov  handle,ax  ;store handle
0036  72 57                            jc   error      ;error return?
```

```
                                ;GET TEXT FROM KEYBOARD
0038  BA 009B R                         mov  dx. offset intro2 ;intro message
003B  B4 09                             mov  ah,9h      ;print message funct
003D  CD 21                             int  21h        ;call DOS
003F                            newline:
003F  BA 0034 R                         mov  dx,offset datbuff ;addr of buffer
0042  B4 0A                             mov  ah,0ah     ;buff kbd input funct
0044  CD 21                             int  21h        ;call DOS
0046  80 3E 0035 R 01                   cmp  datbuff+1,1 ;if no chars typed
004B  7E 37                             jle  exit        :then exit

                                ;insert return and linefeed after typed chars
004D  8A 1E 0035 R                      mov  bl,datbuff+1 ;put character count
0051  B7 00                             mov  bh,0         ; in BX, then insert
0053  C6 87 0036 R 0D                   mov  [datbuff+bx+2],0dh ;return in buf
0058  C6 87 0037 R 0A                   mov  [datbuff+bx+3],0ah ;linefd in buf
005D  80 06 0035 R 02                   add  datbuff+1,2  ;add 2 to count

0062  BA 00B2 R                         mov  dx,offset crlf ;return + linefeed
0065  B4 09                             mov  ah,9h      ;print message functn
0067  CD 21                             int  21h        ;call DOS

                                ;WRITE FILE TO DISK

0069  8B 1E 0086 R                      mov  bx,handle  ;get handle back in BX
006D  BA 0036 R                         mov  dx,offset datbuff+2 ;addr of buff
0070  8A 0E 0035 R                      mov  cl,datbuff+1 ;# of bytes to write
0074  B5 00                             mov  ch,0         ;  into CX
0076  B4 40                             mov  ah,40h     ;write file function
0078  CD 21                             int  21h        ;call DOS
007A  72 13                             jc   error      ;error return?
007C  3A 06 0035 R                      cmp  al,datbuff+1 ;# of bytes written
0080  75 0D                             jne  error        ;same as requested?
0082  EB BB                             jmp  newline    ;go read another line

                                ;CLOSE FILE AND EXIT
0084                            exit:
0084  8B 1E 0086 R                      mov  bx,handle  ;get handle back in BX
0088  B4 3E                             mov  ah,3eh     ;close handle function
008A  CD 21                             int  21h        ;call DOS
008C  72 01                             jc   error      ;error return?
008E  CB                                ret             ;return to DOS

                                ;ERROR ROUTINE
008F                            error:
008F  BA 00AB R                         mov  dx, offset emess ;error message
0092  B4 09                             mov  ah,9h      ;funct # to print mess
0094  CD 21                             int  21h        :call DOS
```

```
ØØ96  8B 1E ØØ86 R                    mov  bx,handle  ;handle is error #
ØØ9A  E8 ØØ9F R                       call binihex    ;print error number
ØØ9D  EB E5                            jmp  exit

ØØ9F                            main   endp            ;end main process
                                ;-----------------------------------------
                                ;
ØØ9F                            binihex proc   near
                                ;
                                ;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
                                ;  TO HEX ON CONSOLE SCREEN
                                ;
ØØ9F  B5 Ø4                            mov    ch,4     ;number of digits
ØØA1  B1 Ø4                     rotate: mov    cl,4     ;set count to 4 bits
ØØA3  D3 C3                            rol    bx,cl    ;left digit to right
ØØA5  8A C3                            mov    al,bl    ;move to DL
ØØA7  24 ØF                            and    al,Øfh   ;mask off left digit
ØØA9  Ø4 3Ø                            add    al,3Øh   ;convert hex to ASCII
ØØAB  3C 3A                            cmp    al,3ah   ;is it > 9 ?
ØØAD  7C Ø2                            jl     printit  ;no, so Ø to 9 digit
ØØAF  Ø4 Ø7                            add    al,7h    ;yes, so A to F digit
ØØB1                            printit:
ØØB1  8A DØ                            mov    dl,al    ;put ASCII char in DL
ØØB3  B4 Ø2                            mov    ah,2     ;display output funct.
ØØB5  CD 21                            int    21h      ;call DOS
ØØB7  FE CD                            dec    ch       ;done 4 digits?
ØØB9  75 E6                            jnz    rotate   ;not yet

ØØBB  C3                               ret             ;done subroutine
                                ;
ØØBC                            binihex endp
                                ;-----------------------------------------
ØØBC                            zread  ends            ;end code segment
                                ;*****************************************
                                       end  start      ;end assembly
```

ZWRITE gets the pathname and turns it into an ASCIIZ string the same way ZOPEN and ZREAD did. However, we're going to assume (as we did with WRITE-F in the last chapter) that we want to create a new file to write to. In this case we need to use the Create a File function, rather than Open a File.

The Create a File Function

There are some important differences between opening a file and creating it (besides the obvious one that an *existing* file is opened while a

new file is created). Most important, it's when a file is created that it can be given an *attribute*, as we'll see below.

Notice that if the file being created has the same name as one that already existed, the old one will be destroyed. Newly created files are always given the read/write access code.

CREATE A FILE Function — Number 3Ch

Enter with:

Reg AH = 3Ch
Reg DS = segment address of ASCIIZ string
Reg DX = offset address of ASCIIZ string
Reg CX = attribute to be given file

Execute:

```
INT 21
```

Return with:

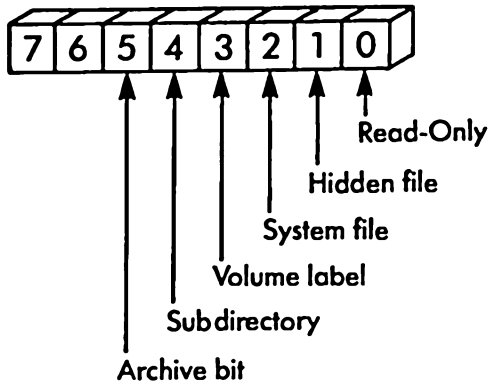
Carry flag = 0 if operation successful
= 1 if error occurred
Reg AX = file handle if file opened successfully
= error code if error occurred

Possible errors are 3, 4, and 5

As the Open a File function did, this function returns the file handle in the AX register.

File Attributes

The *attribute* is a byte permanently assigned to every file to provide information about the file to the operating system. The attribute byte looks like this (notice that more than one of these bits can be set at the same time):



01 — If the Read Only bit is set, then you can't open the file for writing (using function 3D).

02 — Hidden files cannot be seen using DIR.

04 — System files also cannot be seen using DIR. IBMBIO.COM and IBMDOS.COM are marked Read Only, Hidden, and System.

08 — If this bit is set, the file is not a file at all, but the *volume label* for the diskette.

10 — If this bit is set, the file is not a file, but a *subdirectory*.

20 — The Archive bit is set when a file has been written to and closed. It's used by the FDISK utility.

The useful bits for the average programmer are Read Only and Hidden. You can probably think of some files of your own you'd like to keep people from writing on, or knowing about altogether. However, for ordinary files *none* of these bits are set, and the attribute byte is 00. As with file handles themselves, it's not clear why a 16-bit register is used to hold this byte (room for expansion of the byte to a word, perhaps?). The CX register is used to communicate the byte to the Create a File function.

The attribute byte of an existing file can be changed by using the Change File Mode function (43h).

The Write to a File or Device Function

Once the file is created, the ZWRITE program goes on to read a line of text from the keyboard. The *Buffered Keyboard Input* function is used for this in the same way as the disk-writing programs in the last chapter, with the maximum number of characters allowed set to 80d.

Once we've got the line of characters in the DATBUFF buffer, we're ready to write them to the disk.

WRITE TO A FILE OR DEVICE Function Number 40h

Enter with:

Reg AH = 40h
Reg BX = the file handle
Reg CX = number of bytes to write
Reg DS = segment address of data buffer
Reg DX = offset address of data buffer

Execute:

```
INT 21
```

Return with:

Carry flag = 0 if operation successful
= 1 if error occurred
Reg AX = number of bytes actually written
= error code if error occurred

Possible errors are 5 and 6

This function returns, in the AX register, the number of bytes actually written, so your program can compare this with the number of bytes it tried to write. If they're different, there's trouble, probably due to a full disk.

Disk Access Without Records

One of the differences between file handle disk access and earlier forms of disk access is that the Read and Write functions are no longer operating in terms of records.

The ZWRITE program opens the text file on the disk, and then repeatedly gets a string of characters from the keyboard, and writes it to the file. As in the examples of writing to the disk in the last chapter, each new string is appended to the existing file. However, unlike the examples from the last chapter which operated in terms of fixed record lengths, *the strings added to the file don't all need to be the same length.* Each time we call the Write to a File or Device function, we specify the number of bytes to be added on — appended — to the file. We can write 3 bytes to a file one

time, and 2000 (or any other number) the next. They'll just get appended to the file. There's no waste, and no need to fill out unused parts of records with throw-away characters.

In our program, anywhere from 1 to 80 bytes can be appended each time we type in a new line. The program just goes on getting a new line from the keyboard, and writing it to the file, over and over again, until you run out of patience.

The Close File Handle Function

When you type ↵ at the beginning of the line to signify that you're through, the program must then close the file to ensure that the operating system records it on the disk.

CLOSE A FILE HANDLE Function
Number 3Eh

Enter with:

Reg AH = 3Eh

Reg BX = file handle

Execute:

```
INT 21
```

Return with:

Carry flag = 0 if operation successful
= 1 if error occurred

Reg AX = error code if error occurred

Possible error is 6

This is a very simple function, requiring only that the file handle be supplied on entry. The file is closed, and the internal buffer in which DOS has been storing data (waiting for enough characters to fill a 512-byte sector) is written to the disk. You'll see the light go on and hear the disk drive whirr.

Getting to the Middle of a File

In this section we're going to discuss another aspect of file handle

disk access: using files with records. Space does not permit us to show an example of this, but the concept is straightforward, and you should find enough information here to understand how to use the technique.

In the example above we created a text file which consisted of a file of a certain length filled with bytes of text. It was not divided into records, and there was no point in either us, or our program, thinking of it in terms of records.

However, there are situations where a file is more conveniently thought of in terms of records. The birthday files discussed in the last chapter are examples. Other examples would be mailing lists, accounts receivable, and so on — files where each name or customer is given a record containing the same information in the same format.

Now, suppose you have created such a record-oriented file, and you want to access a record in the middle of it, say the 13th record in a 30-record file (perhaps the 13th of your 30 customers). A normal Read will start at the beginning of the file, and a normal Write will start at the end. Assuming we know how many bytes into the file we want to be (probably the number of records times the bytes per record), how do we get there?

Read/Write Pointer

Remember in the ZWRITE program how every time we wrote to the file, the new string was appended to the existing file? The operating system knows where to put the new string because it keeps track of its place in the file with something called the *read/write pointer*. Every time a string was written to the file by ZWRITE, the read/write pointer was set to the end of the string that was written.

To access a particular record in a file we need to move the read/write pointer to the beginning of the record. This is shown in Figure 12-3.

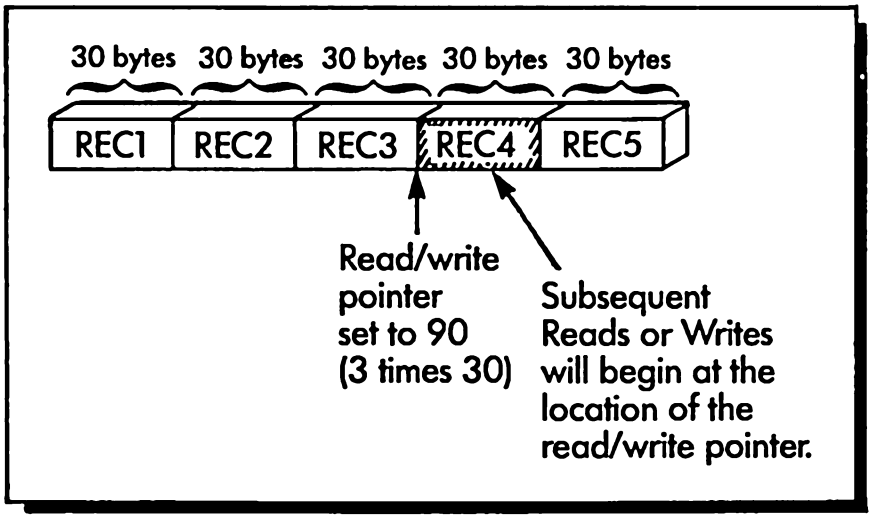


Figure 12-3. The read/write pointer

There is a file handle DOS function which does just what we want: it moves the read/write pointer to any point in the file we specify.

MOVE FILE READ/WRITE POINTER Function Number 42h

Enter with:

Reg AH = 42h
Reg AL = "method value"
Reg BX = the file handle
Reg CX = number of bytes offset into file (high)
Reg DX = number of bytes offset into file (low)

Execute:

```
INT 21
```

Return with:

Carry flag = 0 if operation successful
= 1 if error occurred
Reg DX = new location of read/write pointer (high)
Reg AX = new location of read/write pointer (low)
= error code if error occurred

Possible errors are 1 and 6

The exact operation to be carried out by this function is determined by a number called the "method value." This value is placed in the AL register when the function is called. It can have the following values:

00 — The read/write pointer is moved into the file the number of bytes in the double-word offset CX:DX, counting from the beginning of the file.

01 — The pointer is moved to the current location plus the offset in CX:DX.

02 — The pointer is moved to the end-of-file, plus the offset in CX:DX. When the offset is zero, this returns the file's size.

As we noted, we're not going to provide a program example for this function, but its use is fairly simple. After a file is opened, you use this

function to set the read/write pointer to the location you want in the file. A subsequent Read or Write will start at that point in the file. This gives you the same power to go directly into the middle of a file that random access gave in the last chapter.

Summary

In this chapter you've learned how to read and write files to the disk using the *file handle* method of access. There are many other functions that can be carried out using file handles and ASCIIZ strings. For instance, there are functions which will rename a file (56h) and delete a file (41h). There are also functions which manipulate directories, such as Create a Subdirectory (39h), Remove a Directory Entry (3Ah), and Change the Current Directory (3Bh). While discussion of these functions is beyond the scope of this book, the knowledge of file handle disk access gained from this chapter should make it easy for you to understand these other functions as well. Just read the descriptions in appendix D of the *IBM Personal Computer Disk Operating System* manual.

13

Interfacing to BASIC and Pascal

Concepts

USR function in BASIC
CALL function in BASIC
Memory allocation for BASIC and assembly routines
Use of the BP register
Functions and procedures in Pascal
Use of PUBLIC and EXTERNAL

8088 Instructions

PTR Pointer (pseudo-op)

Applications

DECIHEX — BASIC Decimal to Hexadecimal Converter (uses BINIHEX routine)
HEXIDEC — BASIC Hex to Decimal Converter (uses DECIBIN routine)
COUNTER — Assembly routine to count given letter in string
COUNT — BASIC program to count letters in string (uses COUNTER routine)
PORTIN — Assembly routine to access IN instruction from Pascal
PORTOUT — Assembly routine to access OUT instruction from Pascal
BLAISER — Pascal program to make "blaiser" sounds (uses PORTOUT routine)

Combining assembly-language routines with programs in higher-level languages like BASIC and Pascal is an effective solution to a variety of programming problems. The strong points of assembly language are

its speed and its ability to directly access all of a computer's hardware. Programs in higher-level languages, on the other hand, are often faster and easier to write and debug than assembly-language programs. Many programmers obtain the best of both worlds by writing the main parts of their programs in higher-level languages and coding certain routines — those requiring speed or direct access to various I/O devices — in assembly language.

In this chapter we'll show you how to combine assembly-language routines with programs written in IBM BASIC and IBM Pascal.

General Interfacing Considerations

It is actually fairly easy to interface Pascal to an assembly-language routine. This is because the output of the Pascal compiler is a *machine-language program*, or more specifically an OBJ file. This OBJ file is indistinguishable from an OBJ file produced by the assembler. The

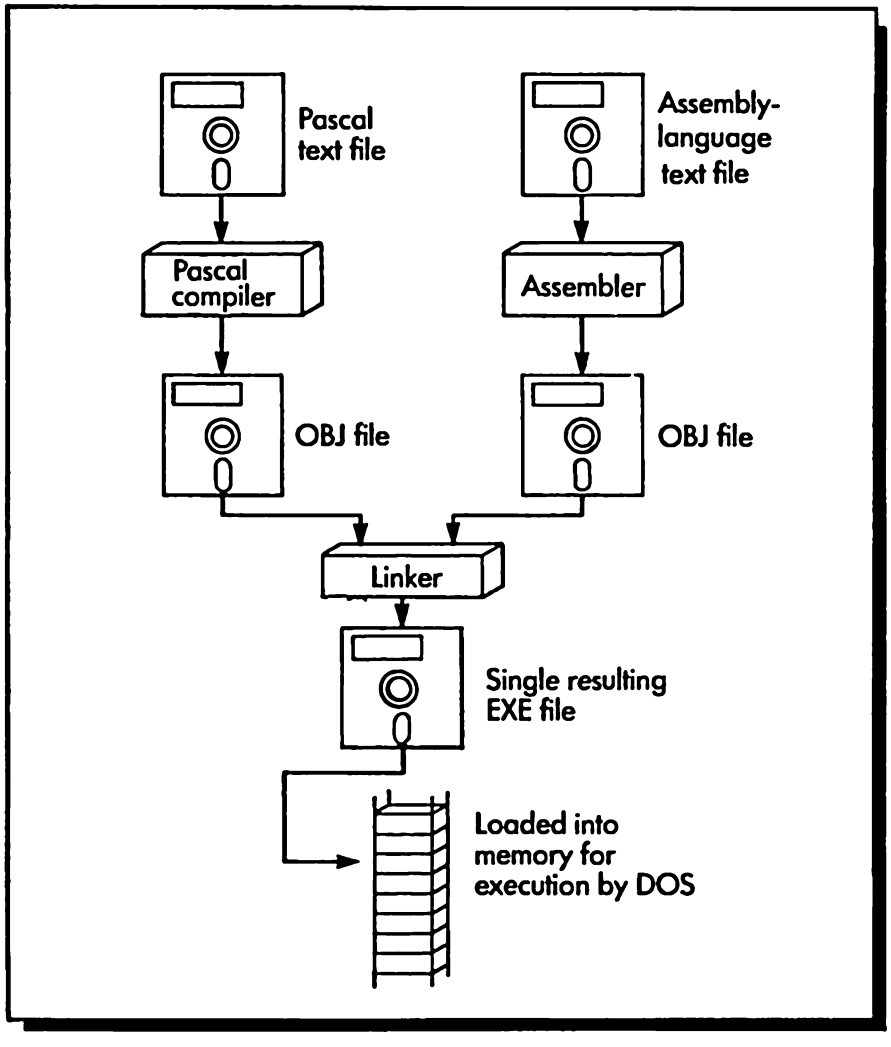


Figure 13-1. Assembly language and Pascal

purpose of the linker program LINK is to merge several OBJ files together to create a single EXE file. LINK can do just this with a Pascal-generated OBJ program and an assembler-generated OBJ routine, and presto, the Pascal program is linked to the assembly-language routine. This is shown schematically in Figure 13-1.

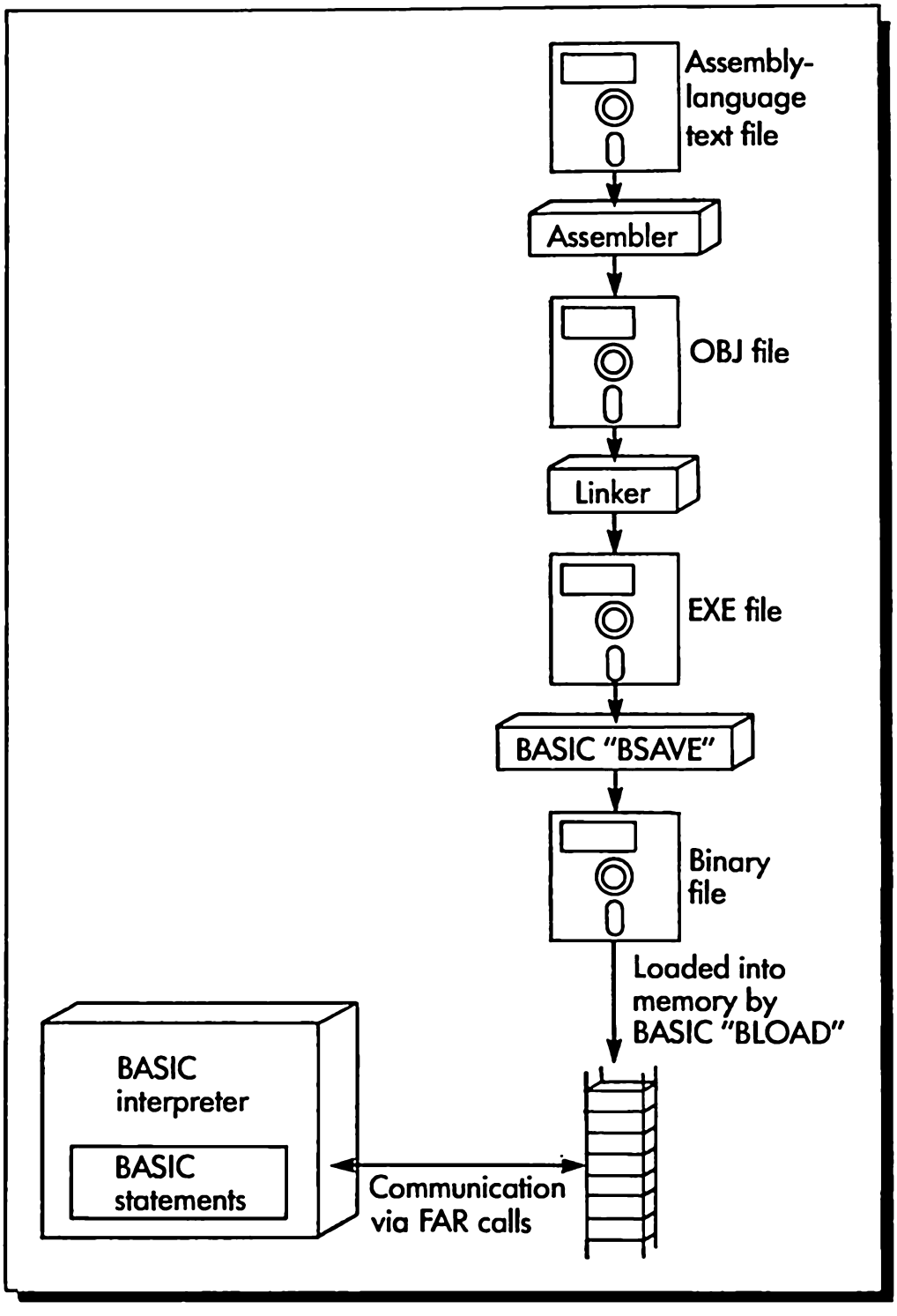


Figure 13-2. Assembly language and BASIC

Other compiled languages, such as C, can be interfaced to assembly-language routines in much the same way Pascal can. In fact, after reading the section on Pascal interfacing, you should be able to combine assembly-language routines with almost any compiled higher-level language.

BASIC, which is an *interpreted* language, is more complex to interface than Pascal. This is because what the *programmer* thinks of as a BASIC program is not the same thing as what the *operating system* thinks of as a BASIC program. To the programmer, the BASIC statements with their line numbers constitute the program; but to the operating system, the BASIC interpreter itself is the program being executed, and the BASIC statements are merely *data* being fed to the program. This is why you can't use the LINK program to connect a BASIC program to an assembly-language routine, or to anything else for that matter. The assembly routine and BASIC interpreter (which contains the BASIC program) simply sit in memory together in different places. It's up to the programmer to know where the assembly routine is located in memory and code this information into the BASIC program. Figure 13-2 shows how this looks.

The Three Parts of the Interface Problem

In the following sections we're going to show you the details of how assembly-language routines are interfaced to programs written in BASIC and in Pascal. Several problems arise when one language is combined with another. We'll mention these problems briefly here, and then — in the separate sections on BASIC and Pascal — show the details of how they are solved in particular circumstances.

Problem 1 — Memory Allocation

Usually when you're running a higher-level language program there's no trouble about where to put the program in memory. The interpreter (in the case of BASIC) or the linker (in the case of Pascal) takes care of figuring out an appropriate place to put the program, and loading it there. You don't usually even need to worry where the program is.

When you add an assembly-language routine to your program, things can get more complicated. In Pascal the process is relatively straightforward: the assembly-language routine is treated as another program module, and the linker decides where it should go. In BASIC, on the other hand, a few tricks are required to figure out how to put the routine in memory, and once it's there to find out where it is so you can tell the BASIC program.

Problem 2 — Transferring Control to the Assembly-Language Routine

Control is generally passed to the assembly routine with a FAR CALL assembly-language instruction executed by the higher-level language. This assembly-language CALL is produced by either a USR or a CALL statement in BASIC, or a CALL instruction in Pascal. The question is: how does the calling program know where the assembly routine is? In BASIC the *programmer* must figure out where the routine is going to be loaded, and code this address explicitly into the BASIC program. In Pascal the *linker* takes care of figuring out the addresses.

Problem 3 — Passing Arguments

In most cases information must be communicated between the assembly-language routine and the higher-level language program. This information is usually in the form of numbers or addresses (which are numbers too). The numbers passed from one program to another are called *arguments*.

The higher-level language program and the assembly routine must agree on a *protocol*, a systematic approach, for passing arguments back and forth. The USR call in BASIC can pass only one argument and return only one argument. These are placed in a special area of the BASIC interpreter called the Floating Point Accumulator, abbreviated FAC. (This is true even for integer arguments.) BASIC's CALL statement, on the other hand, can pass many arguments back and forth between the calling program and the called routine. It works by placing the *addresses of the arguments* on the stack. Pascal's CALL instruction is similar, except that, in addition to placing the addresses of the arguments on the stack, the *arguments themselves* can also be placed on the stack, thus simplifying access.

We'll show the details of all these processes in the following sections.

Interfacing to BASIC with USR

In the following sections we'll assume that you know enough about BASIC programming to follow the rather short programs used as examples. Some of these programs have names that are the same or very similar to programs we've already discussed. These are *new* programs though, so proceed carefully.

The "traditional" (meaning the oldest) way to interface assembly routines to BASIC is with the USR statement. This is a fairly simple

approach which is applicable to routines that don't need more than one argument.

The DECIHEX Program

As our first example of combining an assembly-language routine with a BASIC program with USR, we'll show you a program that translates a decimal number into a hex number. There are really two parts to this operation; first, to get a decimal number from the keyboard and turn it into binary, and then to print out the binary number in hex on the screen. BASIC is very good at the first part: reading a decimal number from the keyboard and turning it into binary. A simple INPUT statement reads the number from the keyboard and stores it as a binary number in BASIC's data space. However, translating binary to hex is not such an easy routine to write in BASIC, since BASIC thinks of numbers in decimal and has no real talent for taking numbers apart bit-by-bit. Here, assembly language is a more natural choice.

Figure 13-3 shows how the two routines relate to each other. Remember: This is a different program than the DECIHEX we saw earlier in the book. We've added the ampersand (&) to the name of the Assembly portion of the program to avoid confusion with the earlier

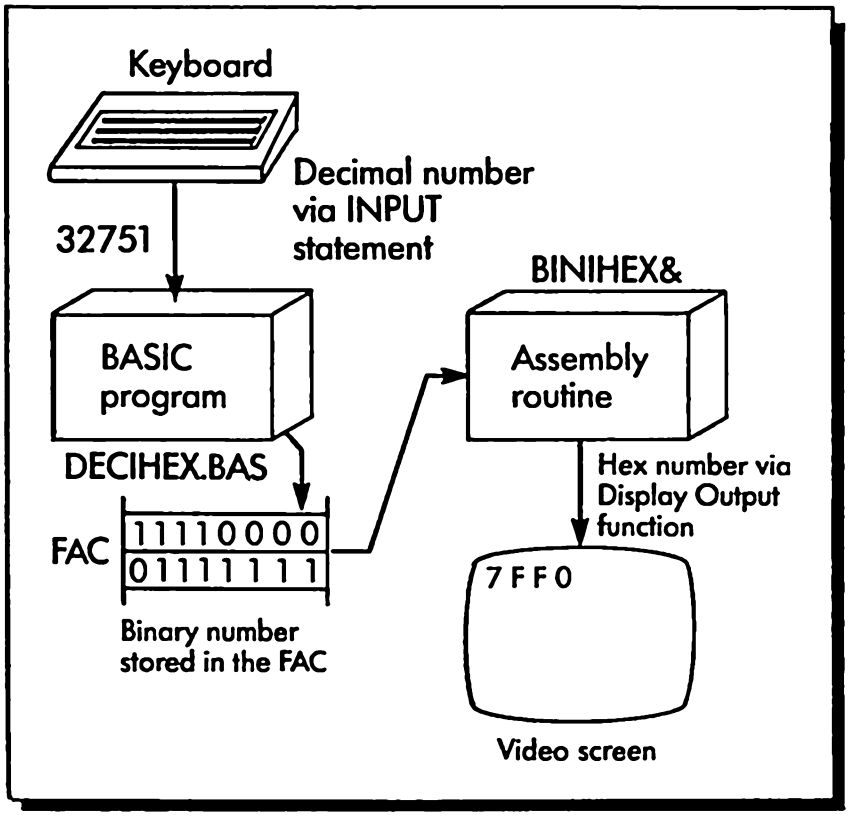


Figure 13-3. Operation of the DECIHEX program

version. Also, we've added ".BAS" to the BASIC part of the program for the same reason. These distinctions will be used throughout the following sections.

The BASIC Program: DECIHEX.BAS

Let's write a BASIC program that will perform its half of the job:

```
1Ø DEFINT A-Z
2Ø DEF SEG = &HØ
3Ø DEF USRØ = Ø
4Ø INPUT "Decimal Number"; N
5Ø PRINT "Hex Equivalent is: ";
6Ø D = USRØ(N)
7Ø PRINT:PRINT
8Ø GOTO 4Ø
```

In line 40 the program invites the user to input the decimal number; the result is stored as variable N.

The key statement in this program is line 60. We want to pass the binary representation of the number N to an assembly routine, and have the assembly routine print out the hex equivalent. Line 60 does just that. Note the argument (N) to be passed to the routine. (You can read all about the USR statement in the *IBM Personal Computer BASIC* manual.) There isn't any value to pass back to BASIC, so D is a "dummy" argument; it's there for format only.

Notice too that we've defined all variables (including N) to be *integers* with the DEFINT A-Z statement. This is important. We must always be very clear what *type* of arguments are being passed.

The Assembly Routine

The assembly routine is very similar to the BINIHEX routine we wrote in chapter 6. However, BINIHEX& will *only* work with BASIC programs; it should not be used with other assembly-language programs.

```
                        ;BINIHEX&--Binary to hexadecimal converter
                        ;  Converts internal binary to hex on screen
                        ;  For use with BASIC programs

= ØØØ2                  display equ     2h      ;video output function
= ØØ21                  doscall equ     21h     ;DOS interrupt number

                        ;*********************************************
ØØØØ                    binihex segment
```

```
                                ;-----------------------------------------
0000                            main    proc    far

                                        assume  cs:binihex

                                ;check that argument is really integer
0000  3C 02                             cmp     al,2    ;2 is code for integer
0002  75 20                             jne     exit    ;wrong argument

                                ;get integer from Floating Point Accumulator
0004  8B 07                             mov     ax,[bx] ;integer into AX
0006  8B D8                             mov     bx,ax   ;into BX for binihex

                                ;print out binary number in BX on screen
0008  B5 04                             mov     ch,4    ;number of digits
000A  B1 04                     rotate: mov     cl,4    ;set count to 4 bits
000C  D3 C3                             rol     bx,cl   ;left digit to right
000E  8A C3                             mov     al,bl   ;move to AL
0010  24 0F                             and     al,0fh  ;mask off left digit
0012  04 30                             add     al,30h  ;convert hex to ASCII
0014  3C 3A                             cmp     al,3ah  ;is it > 9 ?
0016  7C 02                             jl      printit ;no, so 0 to 9 digit
0018  04 07                             add     al,7h   ;yes, so A to F digit
001A                            printit:
001A  8A D0                             mov     dl,al   ;put ASCII char in DL
001C  B4 02                             mov  ah,display ;display output funct
001E  CD 21                             int     doscall ;call DOS
0020  FE CD                             dec     ch      ;done 4 digits?
0022  75 E6                             jnz     rotate  ;not yet
0024                            exit:
0024  CB                                ret             ;return to DOS

0025                            main    endp
                                ;-----------------------------------------
                                ;
0025                            binihex ends
                                ;*****************************************
                                        end     main
```

There are two differences between this routine and the earlier BINIHEX. First, we don't need to save the DS register on the stack. BASIC is not expecting us to do that, the way DOS is. Second, the program starts off with several lines of code which check AL to see what number it contains.

Argument Type Returned in AL Register

Sometimes it's convenient if your assembly routine can have a way of checking that the argument passed to it is what it expected. Other times,

the type of argument may not even be known in advance, and the assembly routine will need a way to figure out what it is. BASIC takes care of this situation by putting a number in the AL register to indicate the type of argument being passed. As an added convenience, this value is the same as the number of *bytes* in the argument:

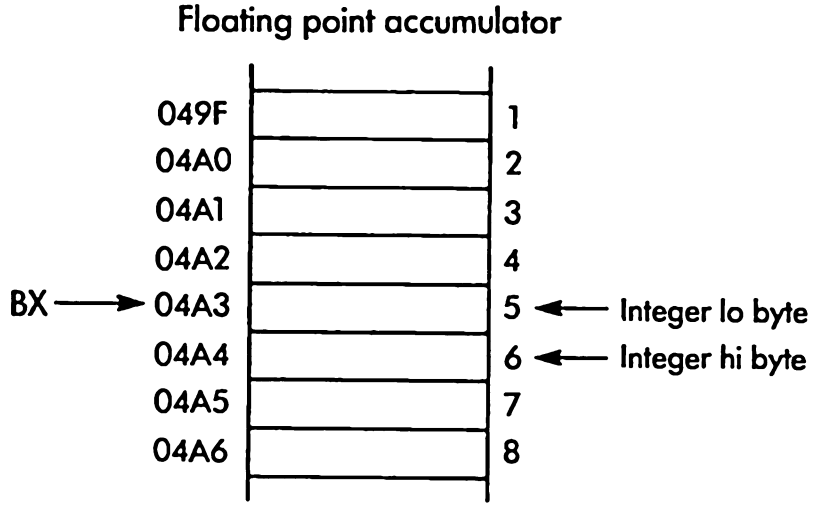
Value in AL Register	Type of Argument
2	Integer (two bytes)
3	String (3-byte string descriptor)
4	Single-Precision Floating Point (four bytes)
8	Double-Precision Floating Point (eight bytes)

Our BINIHEX& program checks that AL contains 2, since it's expecting an integer. If it finds something else, it exits immediately, before it can get into trouble.

The Floating Point Accumulator

The next question is, where is the program going to look for the argument — the binary number N?

In order to carry out floating point operations (those involving arithmetic on single- and double-precision numbers), BASIC uses an area of its internal memory space called the FAC (Floating Point Accumulator). The developers of BASIC decided that this area would also be a convenient place to pass arguments back and forth between BASIC and assembly routines. What does the FAC look like? It's just an 8-byte section of memory, like this:



The argument occupies different positions in the FAC, depending on its type. We've shown in the diagram above how the integer is situated. Single- and double-precision floating point numbers are more complex, and won't be covered here; they are described in appendix C of the *IBM Personal Computer BASIC* manual. Strings are not passed in the FAC at all; we'll show how they're handled later.

Of course, BASIC must tell the assembly routine where in memory its FAC is. It does this by passing the *segment address* of the FAC in the DS register, and the *offset address* in the BX register. This is what the DEF SEG and the DEF USR statements in the BASIC program do: arrange for these values to be placed in the appropriate registers when USR is executed. Actually, it's somewhat more complicated than that. When control is passed to the assembler routine, the BX register always points to *byte number 5* in the FAC, rather than simply to the first byte. Conveniently, this is also where integers are placed.

Notice that at this point we don't yet know what values to use for the segment and offset addresses of the assembly routine, so we don't know what values to use in the DEF SEG and DEF USR statements. Thus the BASIC program DECIHEX.BAS, shown above, has these statements filled in temporarily with zeros. Later we'll see how to find the correct addresses to fill into these statements.

If the assembler routine doesn't need to change the DS register for its own use, DS will retain the segment address of BASIC's FAC throughout the program, as our DECIHEX.BAS program does. If the assembly routine does need to access a data segment of its own, it will need to save DS and restore it when it accesses the FAC and when it returns to the BASIC program.

It should now be clear how the BINIHEX& routine gets hold of the integer it's going to convert to hex. Since BX already points to the integer, the program can simply do an indirect MOV from [BX] into AX.

With the binary value of N firmly in its possession, BINIHEX& can now convert it into hex and print it out in the usual way, using the Display Character function.

Returning to BASIC

The BASIC interpreter originally called the assembly routine with a FAR CALL (generated by the USR statement), so to return to BASIC the assembly routine must execute a FAR RET. Thus (as we learned in chapter 6) the routine must be part of a FAR procedure.

Installing the DECIHEX Program with BINIHEX&

Even though we've finished writing the BASIC program and the

assembly routine, we're still a fair distance from getting them installed in the computer's memory in such a way that they can work together. This process is quite involved, so we'll go through it step by step. (This is a more detailed view of the procedure recommended in appendix C of the *IBM Personal Computer BASIC* manual.)

Deciding Where the Assembly Routine Will Fit

There are a variety of memory locations where the assembly routine can be installed. If you have a 64K system, then the routine must go *inside the area normally occupied by BASIC.* BASIC must then be encouraged not to use this area for itself, since that would destroy the routine. Thus a part of memory above the BASIC work area must be reserved. This can be done either when BASIC is loaded, with the /M option, or from within the program, with the CLEAR command. The following statements both reserve 2K of memory space above the BASIC work area for assembly routines:

```
A>basic /m:&h88ØØ          <— When BASIC is called up

1Ø CLEAR ,&h88ØØ           <— From inside BASIC program
```

Of course, reserving this area decreases the space available for BASIC programs by a corresponding amount. Figure 13-4 shows how the assembly routine is installed inside BASIC's normal space.

For larger systems the assembly routine can be placed *outside* BASIC. This doesn't cost anything in terms of BASIC program capacity. In the examples which follow, we'll assume that you have 96K or more of memory, and can thus place your assembly-language routine outside of BASIC.

Creating an EXE File in High Memory

If the assembly routine is going to go outside of BASIC, then where exactly is it going to go? We can let the linker program figure this out. Until now, we've used LINK to create EXE files in the *low* end of memory. It has installed our EXE files at the lowest available memory address, above the resident part of the operating system. But now BASIC occupies this space. Fortunately LINK has an option which will cause an EXE file to be installed in *high* memory, directly below the transient part of DOS. This is done by appending the parameter "/high" to the program name when we invoke the linker. (We'll show an example of this soon.) Figure 13-5 shows where things will fit together in memory.

Before things get more complicated, let's start making a list of what

has to be done. (You may want to refer to Figure 13-5.) The steps up to this point are:

STEP 1. Create an ASM file of the complete assembly routine.

STEP 2. Assemble it in the usual way.

```
A>asm binihex&
```

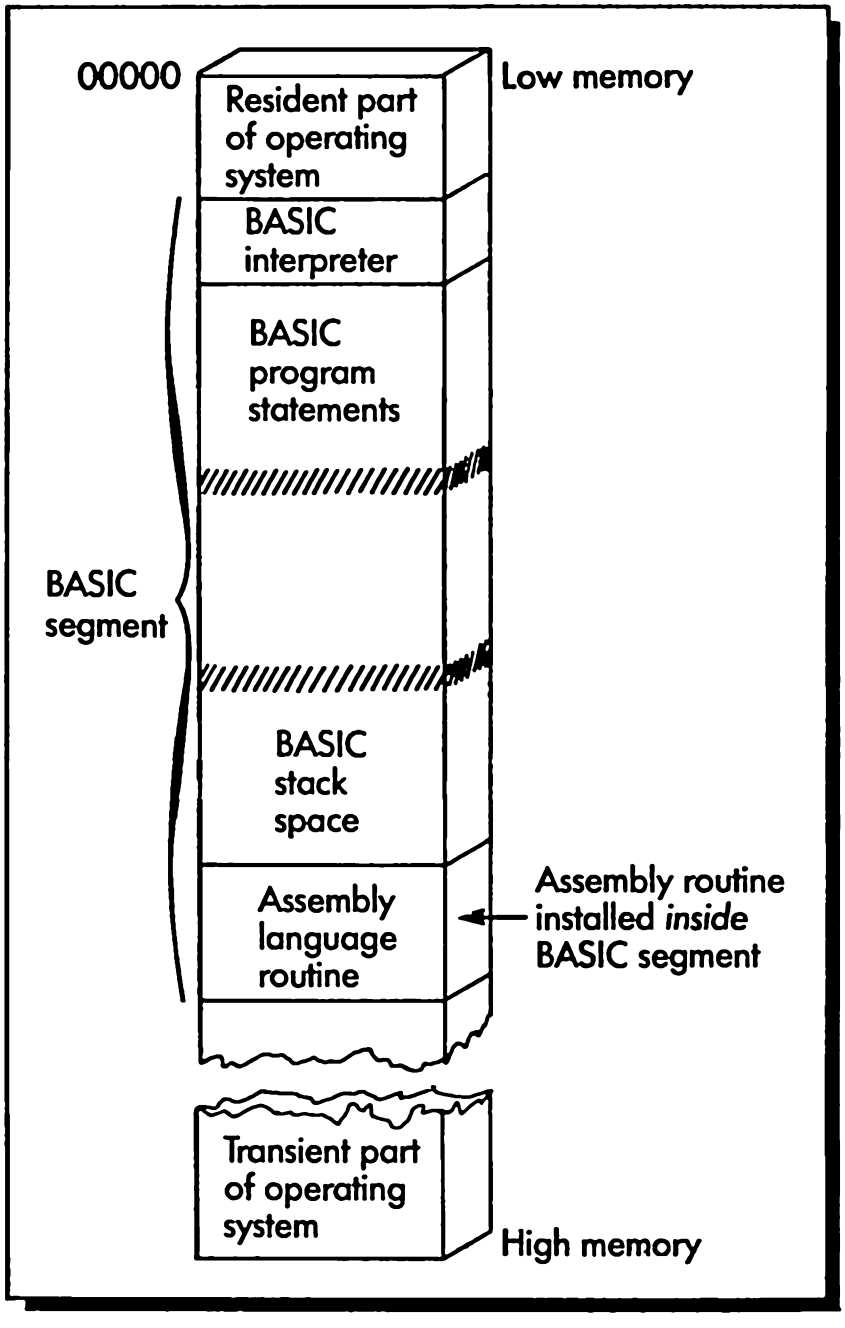


Figure 13-4. Assembly routine inside BASIC segment

STEP 3. Link the resulting OBJ file, using the HIGH parameter.

```
A>link binihex&/high
```

These three steps produce the file BINIHEX&.EXE.

Installing the EXE File with DEBUG

We need to load the resulting EXE file, and find out where it is, so we can tell our BASIC program where to look for it. DEBUG serves the

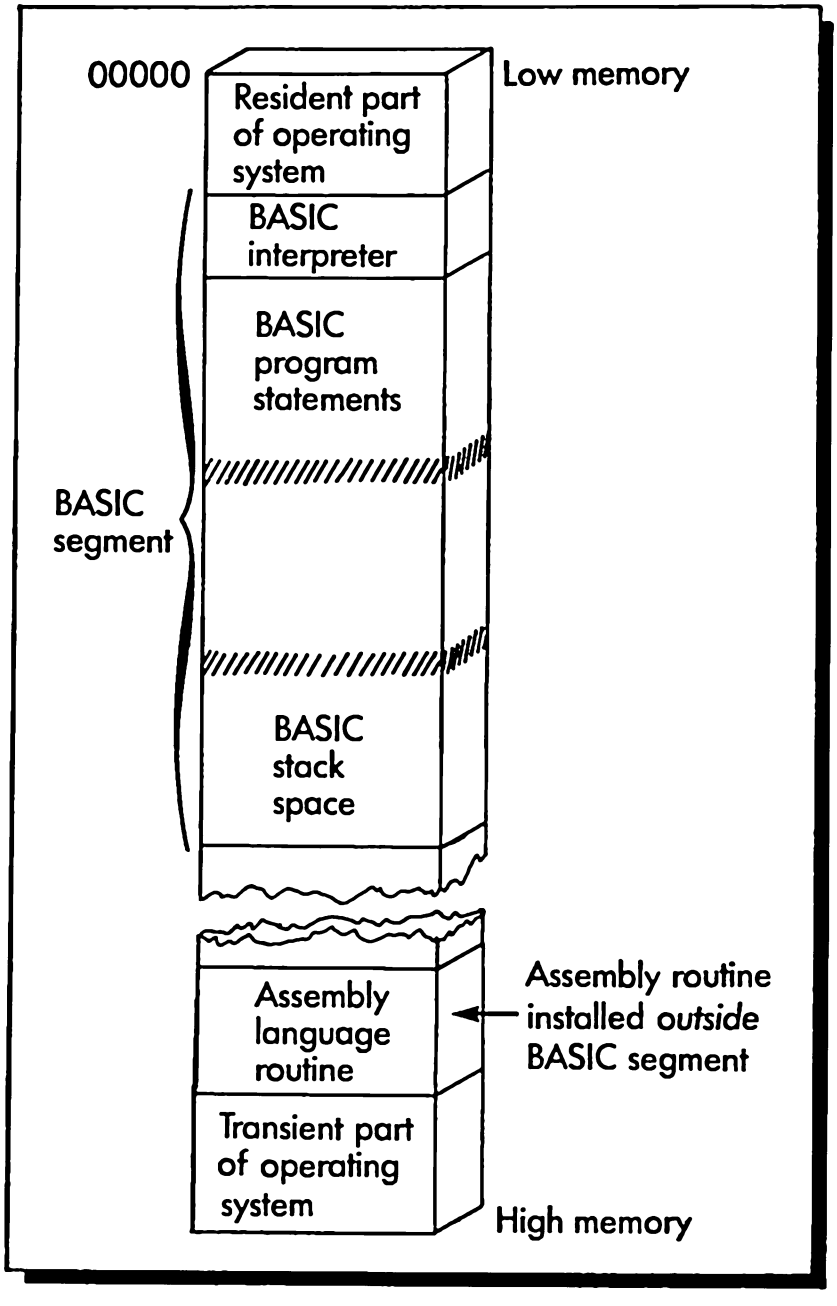


Figure 13-5. Assembly routine outside BASIC segment

purpose here — we call it up with the EXE file of the assembly routine. Then we use the "R" command to look at the registers. The value in the CS register is the *segment address* of our routine; the value in IP is the *offset address*. The offset address will be 0 if a program starts execution at its lowest memory address — that is, if its entry point is 0000 — which is the case in the examples in this chapter.

STEP 4. Call up DEBUG with the EXE file.

```
A>debug binihex&.exe
```

STEP 5. Use "R" to see the values in the CS and IP registers.

```
-r
AX=FFFF  BX=ØØØØ  CX=ØØ8Ø  DX=ØØØØ  SP=ØØØØ  BP=ØØØØ  SI=ØØØØ  DI=ØØØØ
DS=Ø9Ø5  ES=Ø9Ø5  SS=1F94  CS=1F94  IP=ØØØØ   NV UP DI PL NZ NA PO NC
1F94:ØØØØ 3CØ2          CMP     | AL,Ø2    |
                                Segment     Offset
                                address of  address of
                                BINIHEX&    BINIHEX&
```

The next step is really cute. We want to get into BASIC, but we want to do it without losing DEBUG or the assembly routine. So we'll stay in DEBUG, and load BASIC just as if it were any other program, by using the "N" and "L" commands. (We'll load BASICA in the example below, but regular disk BASIC would work just as well.) Once BASIC is loaded, we want to execute it. We can do this with the "G" command.

STEP 6. Set BASIC's name in the FCB for loading.

```
-nbasica.com
```

STEP 7. Load BASIC.

```
-l
```

STEP 8. Execute BASIC.

```
-g
```

At this point the screen should clear and you should get BASIC's

sign-on message and the "Ok" prompt. Now you can either type in the BASIC program DECIHEX.BAS or, if you wrote it earlier and saved it to the disk, you can load it back in with the "load" command.

STEP 9. Load (or type in) the BASIC program.

```
Ok
load"DECIHEX.BAS"
```

Now both the BASIC program and the assembly routine are in memory, and you're ready to tell the BASIC program the segment and offset addresses of the assembly routine, which you learned with the "R" command in step 5. The segment address from the CS register is used in the DEF SEG statement, and the offset address from the IP register is used in the DEF USR statement. In our example, the value in IP was 0, so the value in DEF USR remains 0 as well.

STEP 10. Put the segment address in the DEF SEG statement.

```
2Ø DEF SEG = &H1F94
```

STEP 11. Put the offset address in the DEF USR statement.

```
3Ø DEF USRØ = Ø
```

The resulting DECIHEX.BAS program looks like this:

```
1Ø DEFINT A-Z
2Ø DEF SEG = &H1F94
3Ø DEF USRØ = Ø
4Ø INPUT "Decimal Number": N
5Ø PRINT "Hex Equivalent is: ";
6Ø D = USRØ(N)
7Ø PRINT:PRINT
8Ø GOTO 4Ø
```

You may want to save the BASIC program at this point. That way, if something goes wrong, you won't lose it.

STEP 12. Save the BASIC program.

```
Ok
save"DECIHEX.BAS"
```

The moment of truth is now upon you. The BASIC program is complete, and the assembly routine is installed. Everything should be ready to go.

STEP 13. Execute the BASIC program.

```
Ok
run

Decimal Number? 1Ø          <— You type in the decimal number
Hex Equivalent is: ØØØA     <— The program prints the hex equivalent

Decimal Number? 1ØØ
Hex Equivalent is: ØØ64

Decimal Number? 1ØØØ
Hex Equivalent is: Ø3E8

Decimal Number? 1ØØØØ
Hex Equivalent is: 271Ø

Decimal Number?     <— Here you type Ctrl Break
Break in 4Ø
Ok                  <— You're back in BASIC command mode
```

Debugging

Of course, things may not have gone as smoothly as that. There may be bugs in the assembly routine, in the BASIC program, or in the communications between them. We have several steps still to complete: the saving of the assembly routine as a binary file and the inclusion of a corresponding BLOAD statement in the BASIC program. However, until the program actually works, there's no reason to perform these steps, so let's talk about how to find the bugs in your programs.

We debug the assembly routine by going back to step 4 and starting over at the point where we use DEBUG to load the EXE file. We continue through steps 5, 6, and 7 as before, but when we get to step 8 we do something a little different: we put in a breakpoint (or more than one, if we need to). A convenient place for the breakpoint in a short routine like this is right at the beginning. (Breakpoints were described in chapter 6.) Both the segment and offset addresses must be specified when we set the breakpoint.

STEP 8-A. Execute BASIC with a breakpoint.

```
-g 1f94:Ø          ← Starting address of routine
```

Once BASIC appears we'll load DECIHEX.BAS as we did before in step 9. If DECIHEX.BAS was saved, we probably won't need to do steps 10, 11, and 12 again. We'll simply run the program. As soon as the USR statement is executed, control will go to the assembly routine, and the breakpoint will be triggered, which will return us to DEBUG. From there on we can trace through the assembly routine, examine registers, modify code, and so on, until we find the problem: the usual dark night of the soul.

Saving the Assembly Routine as a Binary File

You could go through all the steps above every time you wanted to load your assembly routine into memory with your BASIC program, but there is a simpler way. Now that you have everything debugged and working, what you want is a way for the BASIC program itself to take over the responsibility of loading the assembly routine.

The problem is that BASIC doesn't understand EXE files. EXE files can be loaded anyplace in memory, depending on where the operating system wants to put them. BASIC, on the other hand, is only comfortable loading assembly-language programs that occupy a definite fixed location in memory. Such files are called "binary files." BASIC can create a binary file with a special command called BSAVE, which means "Binary SAVE."

To use BSAVE your BASIC program needs to know the name of the file you want to save, where it is (both the segment and offset addresses), and how long it is. The segment address must be given with a DEF SEG statement, since BSAVE only gives the offset address. The BSAVE command, with the DEF SEG statement that must accompany it, looks like this:

```
DEF SEG = &h1f94          ← Segment address of file

BSAVE "filename.bin",Ø,&h25
       Filespec      |  Length of file
                     Offset address of file
```

The filename of the program can be whatever you want. The file extension can be BIN to show it's a binary file. The segment address and offset address are the ones you found before in the CS and IP registers,

using the "R" command in DEBUG. The length of the file can be determined from the LST file which you can generate when you assemble the program. It can be expressed in decimal, or in hex if preceded by "&h." (Remember that the first byte is 0, so the number of bytes is one more than the last address used.)

In our case we need to save the BINIHEX& assembly routine as a binary file. Let's add this step to the list:

STEP 14. From BASIC, save the assembly routine as a binary file.

```
Ok
def seg = &h1f94
Ok
bsave"binihex&.bin",Ø,&h25
```

Modifying the BASIC Program to Load the Binary File

Now that the assembly routine has been saved in a form which is palatable to BASIC, we can modify the BASIC program so that it can load the assembly routine itself, thus relieving you of the responsibility. We do this by putting a BLOAD command in the BASIC program. BLOAD is much like BSAVE, except that you don't need to specify the length of the program: this information is in the binary file. Again, the segment address of the file must be specified with a DEF SEG statement. Since this statement is already in your program, only the BLOAD command must be added.

```
DEF SEG = &h1f94          ← Segment address of file

BLOAD "filename.bin",Ø
        |          |
    Filespec   Offset address of file
```

STEP 15. Add a BLOAD command to the BASIC program to load the assembly-language routine.

```
25 bload"binihex&.bin",Ø
```

STEP 16. Save the resulting BASIC program.

```
Ok
save"DECIHEX.BAS"
```

Here's a listing of the resulting complete DECIHEX.BAS program, as it should be recorded on your disk:

```
10 DEFINT A-Z
20 DEF SEG = &H1F94
25 BLOAD"BINIHEX&.BIN",0
30 DEF USR0 = 0
40 INPUT "Decimal Number"; N
50 PRINT "Hex Equivalent is: ";
60 D = USR0(N)
70 PRINT:PRINT
80 GOTO 40
```

Of course the *order* of the statements is important. You can't do the BLOAD until you've defined the segment address with DEF SEG, and you can't call the assembly routine with USR until you've loaded it into memory with BLOAD.

Running the Complete DECIHEX Program

That's about it. After only 16 steps you've got everything organized and ready to go. You can now execute the DECIHEX program from BASIC whenever you want, and it will automatically load in the BINIHEX& assembly routine and interface to it.

Remember, DECIHEX.BAS is not a self-contained program. *The assembly routine BINIHEX&.BIN must be resident as a file on the disk* when you run the BASIC program. In the example above we didn't specify a drive preceding the filespec, so the default or "current" drive is assumed. If the assembly routine will always be on a different drive from the BASIC program, you can use drive specifications (like "b:binihex&.bin") in both the BLOAD and BSAVE commands.

You can use the BINIHEX& routine in any BASIC program. It could be part of a program to simulate a hex calculator, for example, or you could use it in a disassembler program (like the "U" command in DEBUG).

The HEXIDEC Program

In the example above (DECIHEX) we passed an argument *from* the

BASIC program *to* the assembly routine. What happens if we want to pass an argument from the assembly routine back to BASIC?

The next example program, HEXIDEC, takes (as the name implies) a hexadecimal number from the keyboard and prints out the decimal equivalent. The assembly routine HEXIBIN& is used to get the hex number from the keyboard and turn it into binary. This binary number is then placed in the FAC, and control is passed back to the BASIC program which prints out the decimal equivalent of the binary number. (This is the opposite of the previous DECIHEX example.)

We won't repeat all the steps of interfacing the two programs — they are essentially the same as before. Instead we'll concentrate on how the argument is passed from the assembly routine to BASIC.

The BASIC Program

Here's the BASIC program:

```
1Ø DEFINT A-Z
2Ø DEF SEG = &H1F94
3Ø DEF USRØ = Ø
35 BLOAD"HEXIBIN&.BIN"
4Ø PRINT "Hex Number? ";
5Ø N = USRØ(D) : PRINT
6Ø PRINT "Decimal Equivalent is "; N
7Ø PRINT
8Ø GOTO 4Ø
```

As you can see this program has many similarities to the earlier example. However, it now calls the assembly routine first, and then prints out the argument returned, N. In this case the argument passed *to* the assembly routine, D, is a dummy argument.

The Assembly Routine

Here's the listing of the assembly routine HEXIBIN&:

```
                        ;HEXIBIN&--Hexadecimal to binary converter
                        ;  Converts hex from keyboard to binary
                        ;  For use with BASIC programs
                        ;  on entry, BX holds FAC address

= ØØØ1                  key_in  equ     1h      ;keyboard input
= ØØ21                  doscall equ     21h     ;DOS interrupt number

                        ;***********************************************
ØØØØ                    hexibin segment         ;define segment
```

```
                                ;-----------------------------------------
0000                            main    proc    far     ;define procedure

                                        assume  cs:hexibin

                                ;get digit from keyboard, convert to binary
0000  BA 0000                           mov     dx,0    ;clear DX for number
0003                            newchar:
0003  B4 01                             mov   ah,key_in ;keyboard input
0005  CD 21                             int     doscall ;call DOS
0007  2C 30                             sub     al,30h  ;ASCII to binary
0009  7C 18                             jl      exit    ;jump if < 0
000B  3C 0A                             cmp     al,10d  ;is it < 10d ?
000D  7C 0A                             jl      add_to  ;yes, so it's digit

                                ;not dec digit (0 to 9), maybe letter (A to F)
000F  2C 27                             sub     al,27h  ;convert ASCII to bin
0011  3C 0A                             cmp     al,0ah  ;is it < 0a hex?
0013  7C 0E                             jl      exit    ;yes, not letter
0015  3C 10                             cmp     al,10h  ;is it > 0f hex?
0017  7D 0A                             jge     exit    ;yes, not letter

                                ;is hex digit. Add to number in DX
0019                            add_to:
0019  B1 04                             mov     cl,4    ;set shift count
001B  D3 E2                             shl     dx,cl   ;rotate DX 4 bits
001D  B4 00                             mov     ah,0    ;zero out AH
001F  03 D0                             add     dx,ax   ;add digit to number
0021  EB E0                             jmp     newchar ;get next digit

                                ;put number in Floating Point Accumulator
0023                            exit:
0023  89 17                             mov     [bx],dx ;number from DX
0025  CB                                ret             ;return to BASIC

0026                            main    endp            ;end procedure
                                ;-----------------------------------------
0026                            hexibin ends            ;end segment
                                ;*****************************************

                                        end     main    ;end assembly
```

Notice that although the assembly routine doesn't reference the FAC until the end of the routine, it needs to be sure to get the address of the FAC at the beginning of the program and save it, so it will know where it is when the time comes to pass the argument back to BASIC. In this case the problem can be handled very simply: BASIC passes the address of the FAC to the assembly routine in the BX register, which can leave it

there until it needs it. That's why the DX register is used to hold the binary number, rather than BX. If we had been going to use BX in the body of the program, we would have had to save the FAC address somewhere else, probably by PUSHing the BX register onto the stack.

Returning the two-byte integer value from the DX register to the FAC is easy: all it takes is an indirect MOV through [BX]:

```
mov  [bx],dx
```

Escaping from the Program

A small problem arises in this particular program when we want to leave the program and return to the BASIC command mode. Typing Ctrl Break doesn't work! This is because the input from the keyboard to the program happens in the assembly routine, which doesn't know what to do with the Ctrl Break. Actually, if you type Ctrl Break and then follow it with some other character, like a space, you'll find yourself back in BASIC. Another approach would be to modify your assembly routine to recognize Ctrl Break.

Using Strings as Arguments

Numbers, whether they are integers or single- or double-precision floating point, are transferred between BASIC and assembly routines using the Floating Point Accumulator, as we've seen. Strings are handled differently. The reasons for this have to do with the way BASIC stores and operates on strings.

To keep track of all its variables, BASIC maintains a *variable table* which contains all the numeric variables in a given program. BASIC uses this table whenever it needs to find the value of one of the variables. The current values of *numeric* variables are actually kept in the table. However, since strings are long and can be any length, they are not themselves kept in the variable table. Instead, each string is represented in the variable table by a three-byte "string descriptor." The first byte of the descriptor contains the length of the string, and the next two bytes contain the offset address of the string in BASIC's data space. It's this string descriptor that our assembly routine must make contact with if we want to pass string variables back and forth.

Suppose you're using USR to pass control from BASIC to an assembly routine, and you want to pass a string argument to the routine. How do you do it? As far as the BASIC program is concerned, the process is much the same as passing a numeric variable: the string variable is simply listed as the argument in a USR statement. The

assembly routine, on the other hand, no longer looks in the FAC for the variable. Instead, it *looks in the DX register for the address of the string descriptor.* This process is shown in Figure 13-6.

Program to Alter a String

Here's a BASIC program which calls an assembly routine to "code" a particular string into another form. Coding strings in this way can make your programs or data files harder for an unauthorized person to read. To be understood, such strings must be decoded with another program. In this case the coding process consists merely of adding 1 to the ASCII values of each character in the string. The resulting coded string is returned to the BASIC program, and can be printed out.

Here's an example of the program in use:

```
Ok
RUN
Type string to be coded: Now is the time for all good men.
Original version: Now is the time for all good men.
Coded  version:   Opx!jt!uif!ujnf!gps!bmm!hppe!nfo/
Ok
```

Every character is transformed into the one next-higher in the

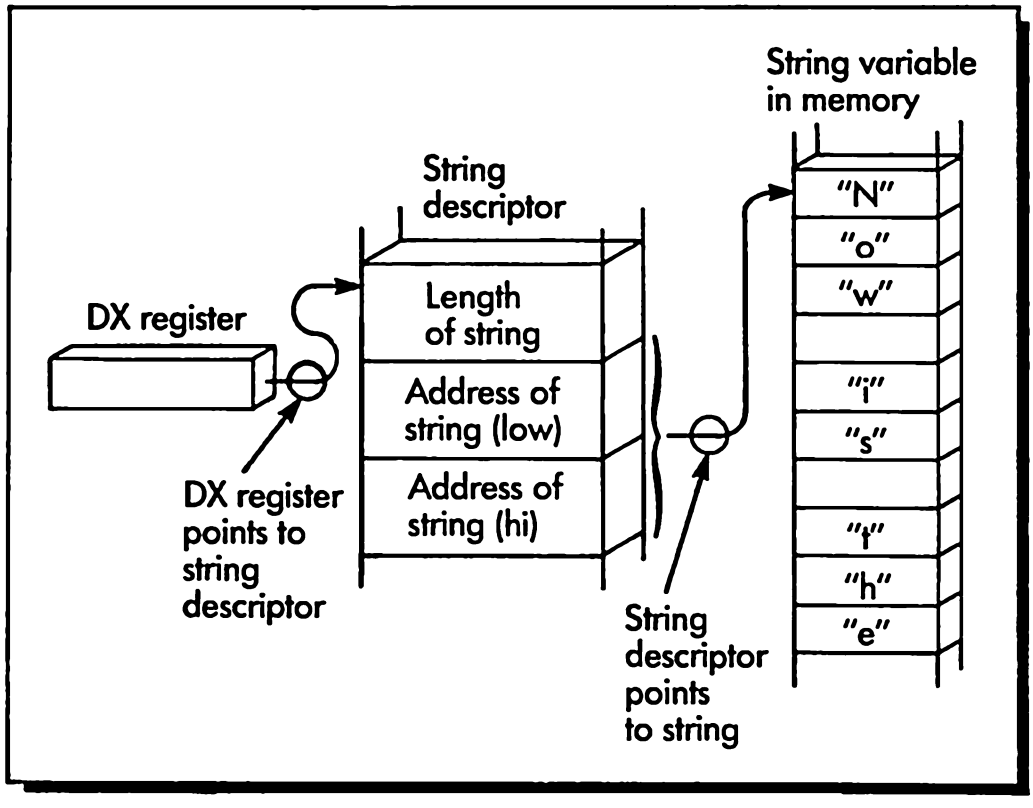


Figure 13-6. Accessing a string descriptor

alphabet: "a" becomes "b" and so on. As a code, this may not stump the KGB for very long, but as soon as you see how to pass string variables you'll be able to write your own more sophisticated coding algorithms. (For instance, you might add 1 to the first letter, 2 to the second, 3 to the third, and so on.)

Here's the BASIC program CODE.BAS:

```
1Ø DEFINT A-Z
2Ø DEF SEG = &H1F94
3Ø DEF USRØ = Ø
35 BLOAD"CODESUB.BIN",Ø
4Ø PRINT"Type string to be coded: ";
5Ø LINE INPUT ST$
6Ø CS$=USRØ(ST$+"")
7Ø PRINT"Original version: "; ST$
8Ø PRINT"Coded  version:   "; CS$
```

There's a rather strange construction in line 60. Perhaps you're wondering why we don't simply say

```
6Ø CS$=USRØ(ST$)
```

The answer has to do with the way BASIC operates on string variables. If we had simply used ST$ as the argument, BASIC would have passed the string descriptor for this variable to the assembly routine, and ST$ itself would have been modified. However, by tacking on the plus sign and quotes, we fool BASIC into thinking we're going to do some arithmetic on the string. It then copies the string into a work area, and does the indicated operation there, which consists of concatenating the null string, which is the same as doing nothing. However, it is now the string descriptor for the *work area* which is passed to the assembly routine. The original string ST$ is left unmodified, as can be seen in the printout of the program's operation above.

Here's the assembly routine CODESUB:

```
                        ;CODESUB--Changes string to "coded" string
                        ;  Adds 1 to every character in the string
                        ;  For use with BASIC programs

                        ;*****************************************
ØØØØ                    codesub segment

                        ;-----------------------------------------
ØØØØ                    main    proc    far
```

```
                                assume  cs:codesub

                        ;check that argument is really a string
0000  3C 03                     cmp     al,3    ;3 is code for string
0002  75 14                     jne     exit    ;wrong argument

                        ;get address of string descriptor from DX
0004  8B DA                     mov     bx,dx   ;descriptor into BX
0006  B5 00                     mov     ch,0    ;clear hi half length
0008  8A 0F                     mov     cl,[bx] ;length of string
000A  83 F9 00                  cmp     cx,0    ;if length is zero,
000D  74 09                     je      exit    ;     then exit
000F  8B 77 01                  mov   si,[bx+1] ;address of string

                        ;add 1 to every character in string
0012                    newchar:
0012  80 04 01                  add  byte ptr [si],1 ;add 1 to char
0015  46                        inc     si      ;point to next char
0016  E2 FA                     loop    newchar ;loop until done
0018                    exit:
0018  CB                        ret             ;return to DOS

0019                    main    endp
                        ;-----------------------------------------
                        ;
0019                    codesub ends
                        ;*****************************************
                                end     main
```

This routine takes the address of the string descriptor out of DX and puts it into BX where it can be used to get the length and address of the string itself through indirect addressing. Since BX is busy holding the address of the descriptor, we'll use the SI register to hold the address of the string. This will be the register which is incremented to point to successive characters in the string as we add 1 to them.

The Pointer Operator: PTR

You'll notice that the instruction that does the adding of the constant 1 to the byte addressed by SI is a rather complex instruction.

```
add  byte ptr [si],1 ;add 1 to char
```

What does "byte ptr" do here? The problem is that if we simply say

```
add [si],1          ;add 1 to char
```

then the assembler will not be clear whether we want to add 1 to the *byte*

pointed to by SI, or to the *word* pointed to by SI. There is no other clue given in the instruction, such as the name of a variable which has already been declared to be a byte or a word.

The PTR operator exists to clarify whether a byte or a word is meant in such situations. It can also be used to override the assembler's assumptions, if it has any, about whether an indirect address refers to a byte or a word.

> An assembly routine must not try to change the length of a BASIC string.

The Returned String Argument

The string passed back to BASIC is simply the modified string. Notice that this string must exist in BASIC before you call the assembly routine, and that the assembly routine *can't change the length* of the string.

Interfacing to BASIC with CALL

The chief limitation of USR is that it allows only one argument to be passed to the assembly routine, and only one to be returned to BASIC. Suppose you need to pass more? You can do it by passing the address of an array or a string in which you've stuffed a number of variables, but IBM's BASIC (unlike earlier BASICs) provides an easier way: the CALL statement.

CALL makes it possible, and even easy, to communicate any number of arguments between BASIC and an assembly routine. In the BASIC program the list of arguments is simply placed in parentheses following "CALL" and the name of the subroutine:

```
CALL SUBR (ARG1, ARG2, ARG3, ARG4, ARG5, etc.)
```

Any of these arguments can go in either direction, that is, from BASIC to the assembly routine or back from the routine to BASIC. The name of the assembly routine to be called, shown as "SUBR" in the example above, is really only the *variable name whose integer value is the offset address of the assembly routine.* Thus the value of this variable must be defined before the CALL statement, as shown here:

```
SUBR = Ø                    ←— 0000 is the offset address of the assembly-language routine

CALL SUBR (ARG1, ARG2)      ←— Name used in call is variable with this value
```

In our case the offset address is 0, so SUBR is simply an integer variable with a value of 0.

What BASIC does is take the addresses of all the arguments in the CALL list and put them on the 8088's stack (which is referenced by the SP register). It's then up to the assembly routine to use these addresses to find the variables it needs for input, and store the variables it wants to send back to BASIC. The tricky part is to come up with an easy way of getting at this list of addresses on the stack. Let's look at an example.

The COUNT Program

In this BASIC program you first enter a string, then you enter a single character. The BASIC program calls an assembly routine called COUNTER, which figures out how many times the character occurs in the string. It passes this number back to BASIC, which prints it out. (You could do the same thing in BASIC with an INSTR statement, but it would be much slower.)

Here's an example of the program in operation:

```
Ok
RUN
String to be searched? She sells sea shells by the seashore.
Character to be counted? s

Character 's' occurs 7 times in string
She sells sea shells by the seashore.
Ok
```

(The capital "S" doesn't count, since the program is rather literal about upper and lower case.) Here's the BASIC part of the program:

```
1Ø DEFINT A-Z
2Ø DEF SEG = &H1F94
25 BLOAD"COUNTER.BIN"
3Ø COUNT = Ø
4Ø INPUT"String to be searched"; S$
5Ø INPUT"Character to be counted"; C$
6Ø AC = ASC(C$)
7Ø CALL COUNT(N,AC,S$) : PRINT
8Ø PRINT "Character '";C$;"' occurs";N;"times in string"
9Ø PRINT S$
```

For simplicity we let the BASIC program convert the search character to its ASCII value, before calling the assembly routine. Thus the string S$ and the integer AC are the values being *passed to the assembly routine,* and the integer N — the count of the number of occurrences of the character in the string — is the value being *returned to BASIC.*

The assembly routine COUNTER looks like this:

```
                        ;COUNTER--Counts number of occurrences of
                        ;             a character in a string
                        ;           For use with BASIC program

                        ;*****************************************
0000                    pro_nam segment para
                                assume  cs:pro_nam
                        ;-----------------------------------------
0000                    counter proc    far

                        ;Assume addresses of three variables are
                        ;on the stack in the form

                        ;0      old BP register     <-- base pointer
                        ;1         hi
                        ;2      offset return address
                        ;3         hi
                        ;4      segment return address
                        ;5         hi
                        ;6      string (lo)     ;bp + 6
                        ;7             (hi)
                        ;8      character (lo) ;bp + 8
                        ;9                (hi)
                        ;10     count (lo)      ;bp + 10
                        ;11           (hi)

0000  55                        push bp         ;save calling prog BP
0001  8B EC                     mov  bp,sp      ;establish new BP

                        ;get address of string descriptor, then
                        ;length of string and address of string
0003  8B 5E 06                  mov  bx,[bp+6]  ;addr of string desc
0006  8A 0F                     mov  cl,[bx]    ;string length in CL
0008  8B 57 01                  mov  dx,[bx+1]  ;string addr in DX

                        ;get addr of character, then character
000B  8B 5E 08                  mov  bx,[bp+8]  ;addr of character
000E  8B 07                     mov  ax,[bx]    ;character in AL
```

```
                                ;search for char in string, count matches
0010  8B DA                          mov  bx,dx      ;addr of string in BX
0012  BA 0000                        mov  dx,0       ;DX holds count
0015  B5 00                          mov  ch,0       ;zero out hi half CX
0017                            next_char:
0017  3A 07                          cmp  al,[bx]    ;match?
0019  75 01                          jne  not_equal  ;no
001B  42                             inc  dx         ;yes, increment count
001C                            not_equal:
001C  43                             inc  bx         ;increment pointer
001D  E2 F8                          loop next_char  ;done string?

                                ;put count into address of count
001F  8B 5E 0A                       mov  bx,[bp+10] ;addr of count in BX
0022  89 17                          mov  [bx],dx    ;put count in address

                                ;restore base pointer, return to BASIC
0024  5D                             pop  bp         ;restore BP reg
0025  CA 0006                        ret  6          ;return, POP 6 bytes

0028                            counter endp            ;end procedure
                                ;-----------------------------------------

0028                            pro_nam ends    ;end of code segment
                                ;*****************************************

                                        end     ;end assembly
```

As you can see, there's something new going on in this program: the use of the BP (Base Pointer) register. Let's digress for a bit and discuss the BP register; then we'll explain how it's used in this program.

The Base Pointer Register

Some registers can be used for indirect addressing, like the BX, SI, and DI registers; others, like AX, can't. Of the registers which can be used for indirect addressing, *each normally operates with a particular segment register.* Thus if you use the BX or SI register indirectly, you can assume that it is referencing the *data segment* (unless you tell them otherwise with a segment override operator). Similarly (as you learned in the section on string-handling instructions) the DI register is assumed to reference addresses in the *extra segment.*

As you've learned, one of the nice things about indirect addressing is that if you have a bunch of data items located next to each other in memory, you can point a register at the first item in the list, and access the others simply by using a different *displacement* in the address. The

displacement is simply the fixed number which is always added to the address in the register. Thus if BX contains the address of the first byte in a list, [bx] will refer to the first byte, [bx+1] will refer to the second byte, [bx+2] to the third, and so on. The numbers 1 and 2 in this example are *displacements*.

Now, when we use CALL to pass variables from BASIC to an assembly routine, it does so by putting the addresses of these variables on the *stack*. There they are, all sitting there next to each other in the stack segment. It would be nice if there were a register which could be used to indirectly address this list of addresses. However, since the stack is (naturally) in the stack segment, none of the registers mentioned above will do the job without additional inelegant finagling. What we need is a register whose natural tendency is to refer to addresses in the *stack segment*.

Why not just use the stack pointer (SP)? Well, if you did, then every time you PUSHed or POPped something in your program, the contents of the SP would change, and all the references to SP would be wrong. What we need is a *second* register which uses the stack segment for its addressing.

Well, it turns out that the Base Pointer register is just what we need; it can be used to indirectly address the stack segment. All the registers which can be used for indirect addressing, and their default segments, are shown in Figure 13-7.

We can put the address of the first item on the stack into the BP, and then, changing the displacement, refer to the items in order as [bp], [bp+2], [bp+4], and so on. The reason it's not [bp], [bp+1], [bp+2], [bp+3] etc., is that only *words* can be PUSHed onto the stack. Since

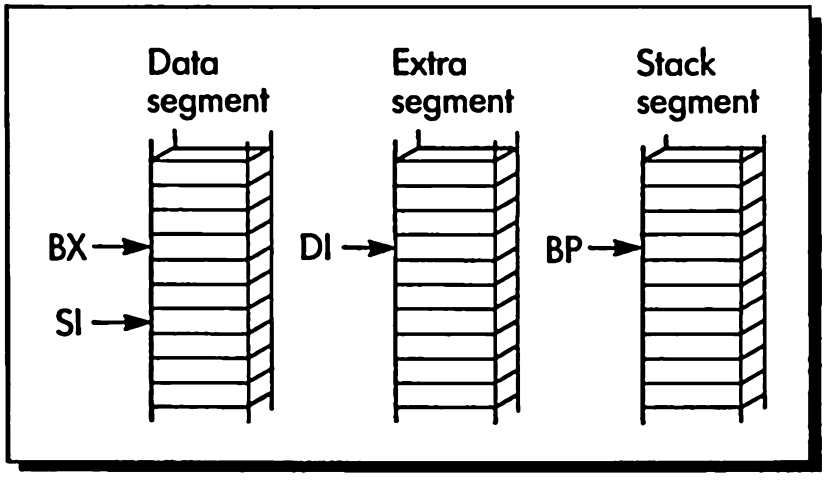
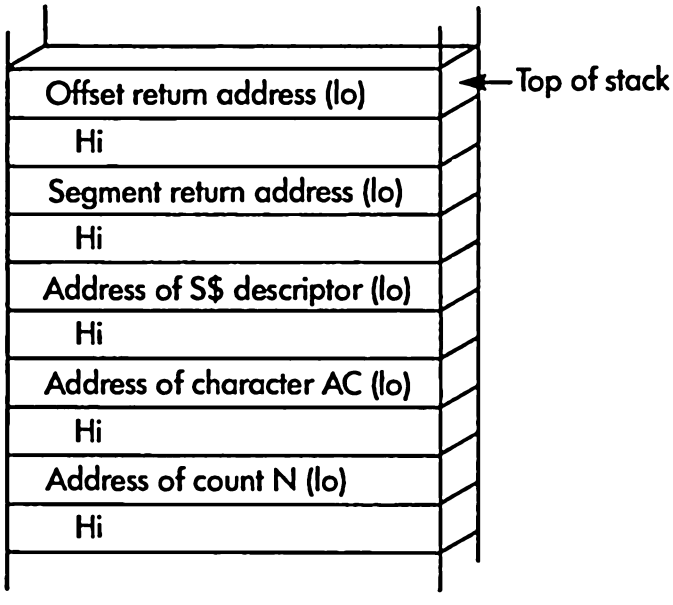


Figure 13-7. Registers and default segments

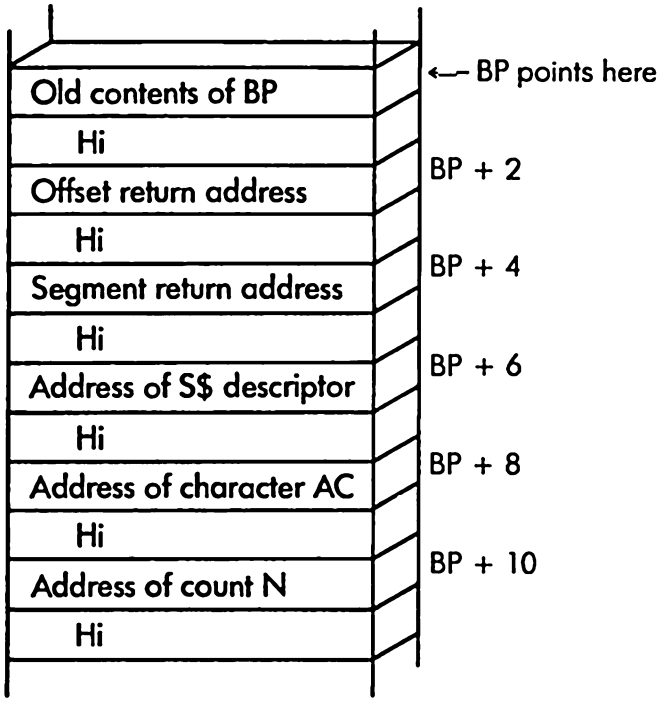
words, rather than bytes, are stored on the stack and referenced by BP, only even-numbered displacements are used with the BP register.

Using BASIC's CALL Statement with the BP Register

In actual practice the arguments aren't at the top of the stack. BASIC first places the addresses of the arguments on the stack, and then does a FAR CALL to get to the assembly routine. A FAR CALL always places the return address (both segment and offset) on the stack. So when control is finally passed to the assembly routine, the stack looks like this:



Since our program is going to be changing the contents of the BP register, it's prudent to save the old value on the stack, in case the calling program was using it. Having done this, we'll then take the current contents of the stack pointer register (SP), and place it in the BP, ready to be used to access the stack. This is done in the first two instructions in the COUNTER program. BP now points to the top of the stack. If we PUSH more things on the stack during the course of our program, SP will change as the top of the stack moves toward the bottom of memory (upwards in the figure), but BP will remain unchanged. The figure below shows where BP points, and how indirect references can now be made to the various items in the stack.



Our program will not be interested in the return addresses; it only cares about the address of the string descriptor for the string S$, the address of the ASCII value of the character to be searched for, and the address of the count to be returned.

The next group of instructions in the program gets the address of the string descriptor from [BP+6], and puts this in BX so it can reference the contents of the descriptor: the length and address of the string. The ASCII value of the character to be searched for is obtained in a similar way, as shown in the code fragment below, excerpted from the COUNTER program:

```
                        ;get address of string descriptor, then
                        ;length of string and address of string
0003  8B 5E 06                mov  bx,[bp+6]  ;addr of string desc
0006  8A 0F                   mov  cl,[bx]    ;string length in CL
0008  8B 57 01                mov  dx,[bx+1]  ;string addr in DX

                        ;get addr of character, then character
000B  8B 5E 08                mov  bx,[bp+8]  ;addr of character
000E  8B 07                   mov  ax,[bx]    ;character in AL
```

Once COUNTER knows where the string and the character are in memory, it counts how many times the character appears in the string. To return the resulting value of N, it simply inserts this integer into the address which it obtains from the stack, as shown in the following instructions:

```
                                ;put count into address of count
001F  8B 5E 0A                        mov  bx,[bp+10] ;addr of count in BX
0022  89 17                           mov  [bx].dx    ;put count in address
```

A FAR RET then returns control to the BASIC program, which finds the value of N waiting for it.

Using CALL Versus USR

As you can see, it's slightly more difficult to set up your assembly routine to use CALL than it is to use USR. CALL, however, provides far more flexibility, in that any number of arguments can be passed both from BASIC to the assembly routine and back from the routine to BASIC.

A possible disadvantage of using CALL is that not all versions of BASIC contain this statement. Thus, if you want your program to be compatible with BASICs running on other computers, you're better off sticking to USR. However, for programs which will only be run on the IBM PC, or on machines using similar versions of Microsoft's BASIC, CALL is a more logical choice because of its superior capabilities.

Interfacing to Pascal

As we hinted at the beginning of this chapter, it's easier to interface an assembly-language routine to a Pascal program than to a BASIC program. In fact, it's so much easier that if you planned to write a large program with many references to assembly routines, it might well pay you to learn Pascal to do it, rather than hassling with the complexities of BASIC.

Let's see what's involved in the assembly/Pascal connection. In the discussion that follows, we'll assume you know enough about Pascal to understand the short programs used as examples.

Pascal uses a very similar technique to the BASIC CALL statement for passing arguments between the program and the assembly routine. That is, the 8088 stack is used to store either the addresses of the arguments or the arguments themselves, and usually (as with BASIC's CALL) the BP register is used as a semipermanent pointer to the stack to facilitate referencing the arguments. (If the use of the stack for this purpose or the use of the BP register is not clear to you, then you should reread the appropriate sections in the first part of this chapter.)

Our example program will demonstrate passing arguments from Pascal to the assembly routine and passing them back again.

The BLAISER Program

BLAISER is a Pascal program which makes the sound of a space-age weapon being fired: the dreaded "blaiser." (Any similarity between the name of this device and the name of a certain 17th century computer hacker is definitely intentional.) A blaiser produces a tone which rapidly shifts its frequency, covering most of the audible spectrum in a half-second or so. Because Pascal is a compiled language (rather than an interpreted one as BASIC is), the code generated when BLAISER is compiled will be plenty fast enough to produce this effect. (BASIC would be too slow in this application.)

Pascal and Sound Generation

To make sounds, a program must be able to access the PC's speaker. Unfortunately, IBM Pascal does not have any statements built into it to produce sound. However, as we saw in chapter 7, the speaker can be activated by IN and OUT assembly-language instructions. Pascal has no such IN and OUT instructions, so if we want to use Pascal to produce sound, we must write an assembly routine containing these instructions, and interface this routine with our Pascal program. In effect we will be extending the Pascal language by adding a function and a procedure to it which will execute the IN and OUT instructions. (Since this example involves the use of the speaker, you should review chapter 7 on sound, if any details of generating sound with IN and OUT instructions are hazy to you.)

BLAISER.PAS: The Pascal Part of the Program

The BLAISER program is borrowed from *Pascal Primer for the IBM PC* by Michael Pardee (New York: Plume/Waite, New American Library, 1984). In that book the Pascal program is divided into two parts. We'll follow the same format here.

Pascal Program Section 1: PORTIO.PAS

The first part of the Pascal program, PORTIO.PAS, consists only of statements that define the assembly-language routines PORTIN and PORTOUT. Separating out this part of the Pascal program permits a variety of different programs to make easy use of these definition statements in PORTIO.PAS, without the statements having to be rewritten for each program.

Here's the listing for this part of the program:

```
{*************************************************************}
PORTIO              Pascal port I/O external declarations
{*************************************************************}
}
Function Portin  (port_addr:word)  :byte;          external;

Procedure Portout (port_addr:word; data:byte);     external;
```

The purpose of these statements is simply to specify the "types" of the function PORTIN and the procedure PORTOUT, and the data types of their arguments; and to declare that PORTIN and PORTOUT are *external* routines. An external routine is one which Pascal will not expect to find when the program is compiled. It is not until the program LINK puts the different assembly language and Pascal routines together that the "external" routines will need to be available as a specified disk file. (We'll explain this process below.)

A *function* is a Pascal construct that returns a value but does not affect anything outside itself, such as variables in memory or I/O hardware. That's what we want PORTIN to do: return the value read from a particular input/output port, without doing anything else. The Pascal program will send this function the value of the port to be accessed, "port_addr" (which is a word type); and the function will return a byte value, "portin", representing the value read from the port.

A *procedure*, on the other hand, can affect things external to itself. In this case our Pascal program will tell the procedure PORTOUT what value it wants to send (the byte type "data") and the port it wants to send it to (the word type "port_addr"), and PORTOUT will transmit the value to the port.

Pascal Program Section 2: BLAISER.PAS

The second part of the Pascal program, BLAISER.PAS, contains the body of the program. An INCLUDE statement in BLAISER.PAS tells the Pascal compiler to include PORTIO.PAS when the program is compiled. This will combine the two sections of the program.

Here's the listing for this part of the program:

```
{
{*************************************************************}
BLAISER.PAS         Blaiser Blast sound demo
{*************************************************************}
}
Program Blaiser_blast (input,output);
```

```
Const
     speaker      = #61;
     timer        = #42;
     toggle_on    = #4f;
     toggle_off   = #4d;
     max_count    = 25Ø;
     scaler       = 2;

Var
     code,
     count        :word

{$include:'portio.pas'}

Begin
  Repeat
    Write ('press enter to fire');
    Readln;
    Portout (speaker, toggle_on);
    For count := Ø to max_count do
       [
       code := Sqr (count) Div scaler;
       Portout (timer, lobyte (code) );
       Portout (timer, hibyte (code) );
       ];
    Portout (speaker, toggle_off);
  Until count = #ffff;
End
```

The values of the various input/output ports which will be accessed are specified in the *declarations* part of the program. Port 61h (called "speaker") is used to turn the sound on, and port 42h (called "timer") receives the number that determines the frequency of the sound. "Toggle_on" and "toggle_off" are the two values which, when sent to port 61h, will turn the timer on and off.

The program lines in the executable part of the program actually generate the sound. First, the program writes a prompt message, "press enter to fire," to the screen, and waits for the user to hit ↵. Once this happens, the program turns the speaker on by sending 4Fh to the speaker timer switch, which is on I/O port number 61h. Then the frequency of the tone is set by sending two bytes in succession to the timer itself, which is port number 42h. These two bytes form a 16-bit number which specifies the frequency of the tone. To produce the desired effect, this tone is then changed and sent to the timer again. This is done *count* number of times. The variable *code* is used as the number to send

to the timer. This number is continuously changed according to the formula

$$code = \frac{count^2}{2}$$

This formula was derived by using trial and error until it sounded right. You can modify it to produce all sorts of different effects.

The program generates the sound of a single blaiser blast, and then displays the prompt again so the user can fire the blaiser over and over by pressing ↵.

PORTIN and PORTOUT: The Assembly Language Part of the Program

Although only one assembly routine, PORTOUT, is actually used by the BLAISER program, two routines are referred to in PORTIO.PAS, so we'll show them both here. You will find them very useful if you're programming in Pascal on the IBM PC and want to generate sounds or use the I/O ports in other ways. (More information on the uses of PORTIN and PORTOUT can be found in *Pascal Primer for the IBM PC*, referenced above.)

Each of these assembly routines is a separate source (ASM) file. They are listed separately below:

```
;***********************************************
PORTIN--Routine to read contents of an I/O Port
;         to be used with Pascal programs

;***********************************************

coder     segment   byte  public ;define segment

          assume    cs:coder

;-----------------------------------------------
portin    proc far        ;define procedure

          push bp         ;save old BP
          mov  bp,sp      ;load current SP into BP

          mov  dx,[bp+6]  ;put port address in DX
          in   al,dx      ;read data byte into AL

          pop  bp         ;restore old BP
          ret  2          ;return to Pascal prog
```

```
portin    endp              ;end procedure
;-----------------------------------------------

coder     ends              ;end segment
;***********************************************
          end               ;end assembly


;***********************************************
PORTOUT--Routine to send data byte to I/O Port
;          to be used with Pascal programs

;***********************************************

coder     segment   byte  public ;define segment

          assume    cs:coder

;-----------------------------------------------
portout   proc far          ;define procedure

          push bp           ;save old BP
          mov  bp,sp        ;load current SP into BP

          mov  dx,[bp+8]    ;put port address in DX
          mov  al,[bp+6]    ;put data byte in AL
          out  dx,al        ;output the byte

          pop  bp           ;restore old BP
          ret  4            ;return to Pascal prog

portout   endp              ;end procedure
;-----------------------------------------------

coder     ends              ;end segment
;***********************************************
          end               ;end assembly
```

PUBLIC Declarations

Notice that in these assembly routines the *code segment* must be declared PUBLIC. When the time comes to link the assembly routines with the Pascal program, this is how LINK will know that PORTIN and PORTOUT will be referenced by another program. Notice too that although it is the *segment* (CODE) which is declared PUBLIC, it's the name of the PROCEDURE (PORTIN or PORTOUT) which will be referenced by the calling program.

Another thing to remember is that the *order* of the various definitions

in the assembly routine is important. Thus the ASSUME pseudo-op must come before the PUBLIC declaration, and so on. If you rearrange things, you may get into trouble.

Passing Arguments and the Stack

How the stack looks when the assembly routine PORTOUT takes control is shown in Figure 13-8.

PORTOUT accesses the information on the stack in much the same way BASIC does, but with one difference. In BASIC, only the *addresses* of the arguments can be placed on the stack. This leaves it to the assembly routine to remove the address from the stack and then use indirect addressing to get at the actual argument. In Pascal, on the other hand, the arguments *themselves* can be placed on the stack and accessed directly by the assembly routine. This is what PORTOUT and PORTIN do.

Passing Addresses

It is also possible in Pascal to pass the *address* of a piece of data, rather than the data itself, although we don't show this in our example. To pass the address of an argument you use the VAR operator in front of the data declaration in the procedure or function definition. For instance, the following statement will cause the address of the DATA1 word-type argument to be passed to the assembly routine:

```
Function Whatsis (var data1:word;) :byte:    external:
```

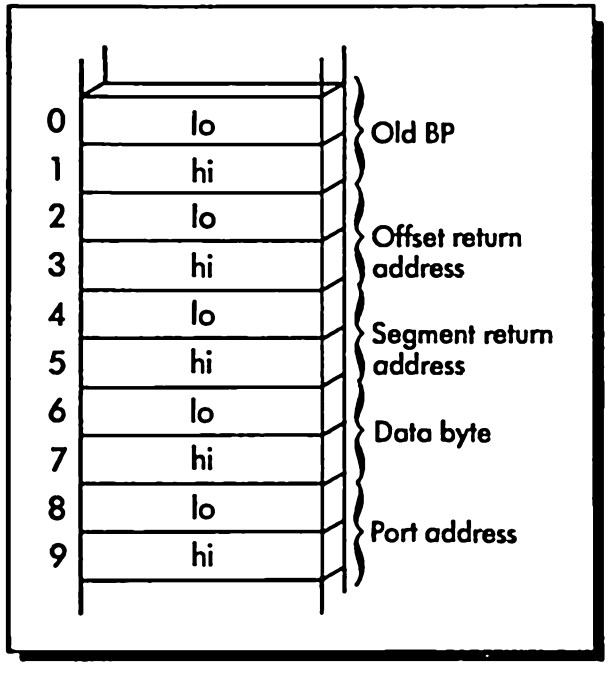


Figure 13-8. The stack when PORTOUT is entered

In this case a byte value is returned by the function, although "data1" is a word variable.

Connecting the Pascal Program and the Assembly Routine

Now that we know what our Pascal program and our assembly routine look like, we can try to put them together into an executable program. For a diagram of this process, refer back to Figure 13-1 at the beginning of this chapter.

Starting with OBJ Files

First we must use the assembler (ASM or MASM) to turn the source (ASM) files for our assembly routines into OBJ files, in the usual way.

```
A>asm portout
```

Then we use the PAS1 and PAS2 parts of the Pascal compiler to compile an OBJ file of the Pascal program. Since the BLAISER.PAS program INCLUDEs the program PORTIO.PAS, it follows that PORTIO.PAS must be available on the disk too.

```
A>pas1 blaiser

A>pas2 blaiser
```

Linking the OBJ Files

Assuming all went well up to now, we're ready to link BLAISER.OBJ and PORTOUT.OBJ together. When we do this we need to be careful that the Pascal library is on the default drive, along with the OBJ files.

```
A>link blaiser+portout
```

The different OBJ files can be listed in the command line, separated by spaces or plus signs.

Running the Program

The result of the linking process is an EXE file, BLAISER.EXE. This can be executed directly from DOS.

```
A>blaiser
```

The result will be the sound of a single blaiser shot.

Summary

In this chapter you've learned how to interface routines written in assembly language to BASIC and Pascal programs. This gives you the valuable ability to rewrite in assembly language those parts of your BASIC or Pascal programs which require speed, bit-twiddling, or access to hardware devices such as Input/Output ports.

You've also learned how to pass arguments from one routine to another by using the stack and the BP register, a technique which can be used not only between assembly language and higher-level languages, but between different assembly-language routines, which can later be linked together with the linker.

Appendix A — Hexadecimal Numbering

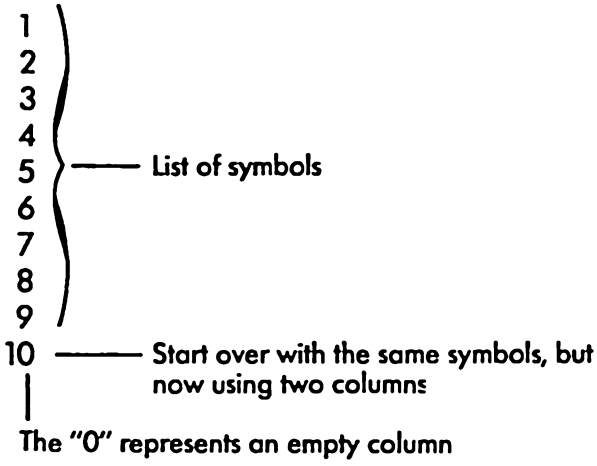
In this appendix we're going to discuss the hexadecimal numbering system. Familiarity with this system is essential if you want to write assembly-language programs, and it's also useful in other areas of the computer field — understanding it is a real mark of computer literacy. Unfortunately, like assembly-language programming itself, hexadecimal numbers can look intimidating at first, with their strange mixture of letters and decimal digits. We hope the following discussion will demystify the hexadecimal system for you, and thus provide a valuable tool for understanding programming and computers.

What Is a Numbering System?

Over the course of millennia humans have learned to assign symbols to different numbers of objects. At first these symbols were oral: "*one* mastodon, *two* clubs, *three* men." When writing came into use these counting symbols were translated into a written form: one wedge-shaped symbol meant "one," two such symbols together meant "two," and so on. This was all right for small numbers of objects, but writing fifteen little wedges to stand for fifteen sheep was inconvenient.

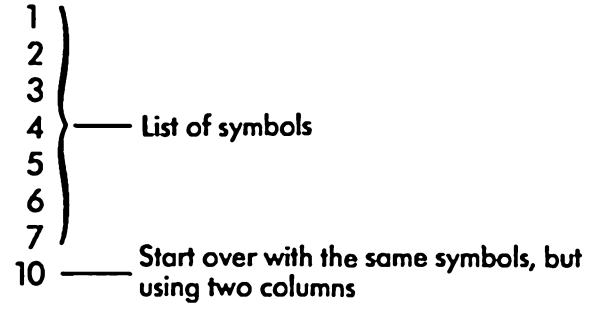
The Roman numbering system was an attempt to streamline things by assigning single symbols to numbers other than one, so that "V" was five, "X" was ten, and so on. However, this system was not completely successful, as generations of school children can attest.

It was the Arabs who figured out a system so efficient that it is still in use today. Their idea was to assign single symbols to numbers up to a certain value, and then *start over in a different column* when the list of symbols had been exhausted, using a special symbol to indicate an "empty" column. Thus,



This system seems perfectly natural to us, since we are so used to working with it; but in fact the idea of using *columns* in this way — to indicate value—and the idea of using a symbol to stand for nothing, or "zero," are both very profound. It's hard to imagine a successful numbering system that does not use these concepts.

Notice, however, that the number of symbols to be counted before moving to the next column is purely arbitrary. There is no particular reason why there are exactly ten numbers. Well, of course there *is* a reason: the fact that we happen to have ten fingers. But there's no *mathematical* reason. Counting, arithmetic, and mathematics would all work just as well with some other number, say eight:

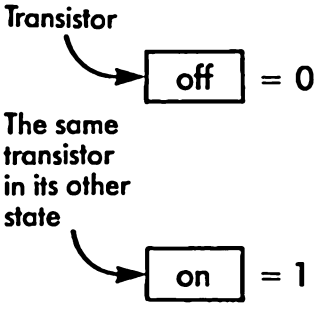


In fact, several numbering systems besides those based on ten have been used in the course of history. The Babylonians favored a system based on sixty, which lingers on, anachronistically, in our clocks and watches. In this system we count up to fifty-nine seconds, then increment the next column to one minute and start over with zero seconds.

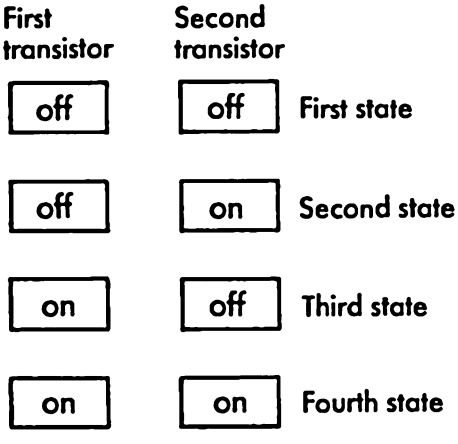
What Numbering System Do Computers Like?

If people feel at home with the *decimal* numbering system (ten numbers) because they have ten fingers, what numbering system do

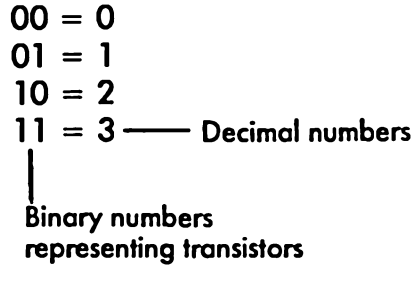
computers feel at home with? Computers are filled with thousands of tiny switches called transistors. Each of these little devices can be in either of two states: switched on or switched off. That is, a transistor can store this very small amount of information: "on" or "off." This much information — the choice between two things (yes or no, on or off, black or white) — is called a *bit*. If we decide to call the "off" state of a transistor "zero," and the "on" state "one," we have a very simple numbering system with only two possible states.



To represent more than two numbers in a computer we need more transistors. Suppose, for example, we have two transistors. Each can be either 0 or 1, so together they can represent four different states:

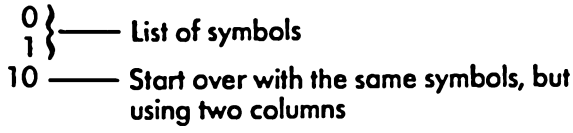


Let's simplify how we write this by representing the little transistors more symbolically, using "0" to stand for "off," and "1" to stand for "on."

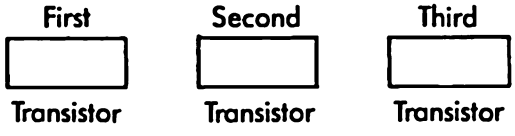


The Binary System

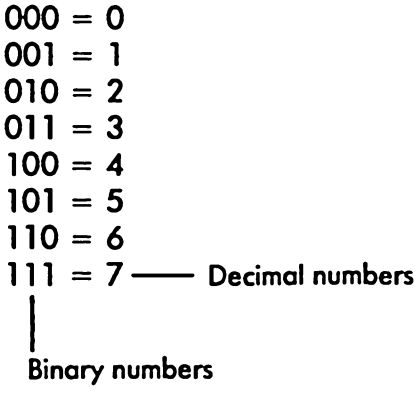
The numbers on the left, which stand for the "on" or "off" state of the transistors, are examples of the *binary* numbering system. *Binary* means "based on two," just as *decimal* means "based on ten." Notice how, since there are only two numbers in binary, 0 and 1, we must put a 1 in the next column over after only two things have been counted:



Suppose we had three transistors:



How many things could we count? Let's represent the transistors in binary again:



As you can see, each time we add a transistor — which is the same thing as adding another column in our binary numbering system — we can count up twice as far as we could before. With four transistors, or binary digits, we can count to 16; with five we can count to 32, and so on.

Computers frequently use eight transistors — which is the same as eight bits — to represent a number, so the number can be as large as 255, as shown in this table:

```
00000000 = 0
00000001 = 1
00000010 = 2
00000011 = 3
00000100 = 4
00000101 = 5
00000110 = 6
00000111 = 7
00001000 = 8
00001001 = 9
00001010 = 10
00001011 = 11
00001100 = 12
00001101 = 13
00001110 = 14
00001111 = 15
00010000 = 16
       .
       .
       .
11111100 = 252
11111101 = 253
11111110 = 254
11111111 = 255 ——— Decimal numbers
       |
       |
       Binary numbers
```

From the above discussion we can see that there are two numbering systems in the computer world: the binary numbering system, which is natural for computers, and the decimal system, which is more natural for humans.

How much of a problem is the use of these two numbering systems — decimal and binary — in assembly-language programming? If we need to convert from binary to decimal, we can use a table like the one above, or better yet, let the computer figure out what the decimal equivalent of a particular binary number is, and print it out. Isn't this all we need to know? After all, most higher-level computer languages, such as BASIC, do this sort of conversion so routinely that we're not even aware that the computer is thinking in binary: it prints out decimal numbers, we type in decimal numbers, and the computer takes care of all the conversions. Why can't we do the same thing in assembly language?

The problem is that in assembly language (and in various non-

standard circumstances in other languages as well) we often need to look at the data in the computer in its untranslated or binary state. This is important because we're often concerned with the state of particular *bits*, rather than with the numbers these bits represent. When we're interested in bits we want a numbering system that shows us the state of these bits — on or off, 0 or 1. We can, for instance, immediately see that the binary number 11111100 has all its bits set to 1, except the two on the right; whereas when we look at the equivalent decimal number, 252, this information is no longer obvious.

The reason the decimal number 252 doesn't tell us very much about bit patterns is that each decimal digit does not represent a fixed number of transistors (or binary digits). For instance, you can't use exactly three binary digits to represent the decimal numbers, since with three bits — as we saw above — you can only count up to eight. On the other hand, if you use *four* binary digits you're forced to count up past ten to sixteen. One decimal digit represents somewhere between three and four bits: so there just is no simple relationship between binary and decimal numbers.

The Hexadecimal System

What would be ideal for talking to computers is a numbering system that has the advantages of binary — an easy to understand relationship between the state of the transistors in the computer and the number itself — and of decimal — numbers concise enough to be easily understood by humans.

Two such systems are in fairly wide use: the *octal*, or base-eight system, and the *hexadecimal*, or base sixteen system. Octal is actually much easier to learn than hexadecimal, but it takes three binary bits to represent an octal number, and three is thought to be an awkward number in the computer business, since it does not go evenly into an 8-bit byte. Hexadecimal has therefore become the standard computer numbering system.

What exactly is a "base-sixteen" system? It has sixteen symbols for numbers, starting at 0 and going up to . . . oops. We run out of decimal digits at nine, so we need six more. What to do? Why not use some other common symbols — letters — for the digits beyond nine? The result looks like this:

```
 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 =  A = 10
 1011 =  B = 11
 1100 =  C = 12
 1101 =  D = 13
 1110 =  E = 14
 1111 =  F = 15
10000 = 10 = 16
10001 = 11 = 17
10010 = 12 = 18
    .    .    .
    .    .    .
    |    |    |
    |    |    Decimal numbers
    |    Hexadecimal numbers
    Binary numbers
```

Notice how four binary digits correspond to exactly one hexadecimal digit. When the hexadecimal number gets so big it has to use *two* digits (going from F to 10 hexadecimal, which is from 15 to 16 decimal), the binary numbers also shift into another column (from 1111 to 10000). It's this exact relationship of four binary digits to one hexadecimal digit that makes the hexadecimal numbering system so much more useful in computers than the decimal system.

Hexadecimal to Binary Conversions

When you see a hexadecimal digit — in a DEBUG dump, for instance — you can convert it immediately to binary, using the table above.

If there are *two* hexadecimal digits in a number, they are converted to binary one at a time, again according to the above table. For instance,

```
A8h = 10101000
      |  |
      A  8
```

since Ah = 1010, and 8h = 1000. (From now on in this appendix, numbers followed by "h" will represent hexadecimal numbers.)

Hexadecimal numbers with any number of digits can be converted to binary in a similar way. For instance,

B49Ah = 1011010010011010

Hexadecimal Arithmetic

What happens when you try to perform arithmetic in the hexadecimal numbering system? For small numbers it's not so hard. For instance,

```
   4h
+  2h
----
   6h
```

This is just the same as decimal.

How about

```
   Ah
+  4h
----
   Eh
```

Not too bad either. We count 4 past A: "B, C, D, E," much as we used to count on our fingers when we were first learning the decimal system.

When we need to *carry*, things get a little trickier, since we need to remember that "F" in hexadecimal plays the role of "9" in decimal: it's the last digit before 10.

```
   Ah
+  8h
----
  12h
```

We find this result by counting eight digits past A: "B, C, D, E, F, 10, 11, 12."

After a while you get the hang of doing hexadecimal arithmetic on small numbers. However, large numbers are another story. Confronted with

```
  A84Bh
+ 7C5Fh
-------
    ?
```

most of us would head for the showers. What to do? One answer is to convert the hexadecimal numbers to decimal, do the arithmetic, and convert them back again to hexadecimal.

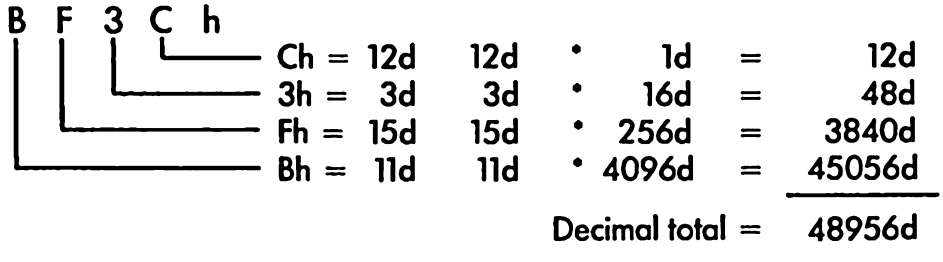
Converting Between Hexadecimal and Decimal

Besides simplifying arithmetic on large hexadecimal numbers, hexadecimal to decimal conversions are often useful in their own right. For instance, to use PEEKs and POKEs in BASIC, you may need to represent memory addresses in decimal; whereas technical literature on computers often gives memory addresses in hexadecimal. Assuming you don't have a conversion table, you'll need to do the conversion by hand.

Hexadecimal to Decimal Conversions

The important thing when finding the decimal equivalent of a hexadecimal number is to remember that the digits in each column of a hexadecimal number are each worth sixteen times more than the digits in the column to the right. Thus 30h is sixteen times larger than 3h, and 300h is sixteen times larger than 30h. (In the same way digits in the ten's column in the decimal system are worth ten times more than the digits in the one's column, and so forth.)

Let's find the decimal equivalent of BF3Ch. (In these examples we'll show decimal numbers followed by "d" to avoid any possibility of confusion.) The one's column of the hexadecimal number is easy: we simply look up the number in the table above, which tells us that Ch is 12d. The ten's column must be multiplied by sixteen, the hundred's column must be multiplied by 256d (which is 16d times 16d), and the thousand's column by 4096d (256d times 16d).



Thus BF3C hexadecimal is 48956 decimal.

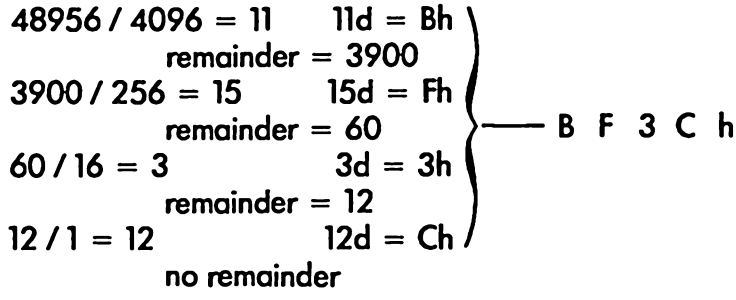
Decimal to Hexadecimal Conversions

To do conversions in the other direction, from decimal to hexadecimal, we reverse the process: we now use division by 16 instead of multiplication. Let's take the same number we just converted to decimal

and see if we can convert it back to its original hexadecimal value.

To find the hex digit on the left (the "thousands" column), we want to see how many times 4096d goes into the number. For the hundred's column we want to see how many times 256d goes into the remainder of the preceding thousands operation, and so on.

Here's what the complete process looks like:



That's really all there is to hexadecimal to decimal conversion. The methods shown work for numbers with any number of digits — the more digits, the more steps are required, but the process is the same.

Now you understand binary numbers and hexadecimal numbers, and how to convert back and forth between these new systems and the old familiar decimal system. As with most new skills, practice is the best way to increase your familiarity with hexadecimal. Keep plugging away, and eventually those funny numbers will start to seem perfectly normal, and you'll wonder why all humans, or at least programmers, don't grow more fingers and count in hexadecimal too!

Appendix B — Supplementary Programs

Programs

MEMSCAN — Displays usage of the PC's address space
HEXIDEC — Hexadecimal to decimal converter
PRIME — Finds prime numbers using sieve of Eratosthenes
SET-BD — Creates a file of birthdays
GET-BD — Lists people with birthdays on today's date
MOD-BD — Modifies the birthday file
SAVEIMAG — Saves screen image to disk file

This appendix contains listings for a number of programs which are too long to include in the main part of the book. These programs offer additional examples of assembly-language programming, while at the same time providing some interesting and useful utilities for your IBM PC system.

Only the ASM files for these programs are listed. They should be typed in as shown, then assembled and linked into EXE files.

MEMSCAN

MEMSCAN produces a visual image of the entire one-megabyte address space of the PC. Those areas of memory which are currently in use (occupied by programs) are marked with Xs, while unused areas are filled in with periods. This can be useful in figuring out which areas of memory are occupied by the operating system, which by applications programs, how large a particular program is, and so on. To use this program, simply enter "memscan" following the DOS (or DEBUG) prompt, as shown below:

```
A>memscan

      0000     2000     4000     6000     8000     A000     C000     E000
0000: XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXX.  XX.XXXX
1000: XXXXXXXX XXX..... ..XXXXX X....... ....X... ........ ..XXXXX XXXXXXX
2000: ........ ........ ........ ........ ........ ........ ........ ........
3000: ........ ........ ........ ........ ........ ........ ........ ........
4000: ........ ........ ........ ........ ........ ........ ........ ........
5000: ........ ........ ........ ........ ........ ........ ........ ........
6000: ........ ........ ........ ........ ........ ........ ........ ........
7000: ........ ........ ........ ........ ........ ........ ........ ........
8000: ........ ........ ........ ........ ........ ........ ........ ........
9000: ........ ........ ........ ........ ........ ........ ........ ........
A000: ........ ........ ........ ........ ........ ........ ........ ........
B000: XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX ........ ........ ........ ........
C000: ........ ........ ........ ........ ........ ........ ........ ........
D000: ........ ........ ........ ........ ........ ........ ........ ........
E000: ........ ........ ........ ........ ........ ........ ........ ........
F000: XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX XXXXXXXX

X    X    X    X    X    X    X    X
000  400  800  C00  1000 1400 1800 1C00
```

Segment addresses are shown on the left-hand side, and offset addresses run across the top. Each individual symbol (an X or a period) represents 400h bytes of memory. The values of individual symbols in each group of eight positions are shown for reference at the bottom of the printout.

Notice that in this example the monochrome display can be seen to be in use at B000:0 to B000:7FFF. The color display memory (at B800:) is not in use. The resident part of the operating system and various other programs occupy the lower part of memory, and the transient part of the operating system can be seen starting at 1000:C800. Since this is a 128K (20000h bytes) system, there is nothing above 1000:FFFF. Isolated areas of usage with a single "X", such as the one at 1000:9000, are probably stack areas.

To see only the area occupied by the operating system, power down the computer to erase all the memory locations. Then boot up the operating system and immediately run MEMSCAN. The resulting Xs will show you how big your operating system is. You can also see how large other programs are by loading them, then exiting them and running MEMSCAN. MEMSCAN will overlay the previous program, but that

won't matter, since the memory locations occupied by a program will not be erased until the computer is powered down.

MEMSCAN works by making certain assumptions. First, it assumes that memory with nothing in it will have all the bytes set to the same value. (This could be 00, or FF, or some other value, depending on the hardware.) Second, it assumes that memory with a *program* in it will have bytes set to very different values. MEMSCAN examines pairs of memory locations. If they are different, there is a possibility that a program occupies that part of memory. To be sure, the program checks a third location. If it's different too, the program marks that 400h-byte section with an X and goes on to the next section. If a pair of locations is the same, the program skips over eight bytes and looks again.

Here's the listing for MEMSCAN:

```
;MEMSCAN--Scans entire megabyte of memory
;
;  Assigns a symbol to each group of 1Ø24 bytes
;    symbol is "." if nothing there
;    symbol is "X" if something there
;
;  Output arranged:
;    8 groups/division,
;    8 divisions/line,
;    16 lines/screen
;
;    1Ø24 * 8 * 8 * 16 = 1Ø48576
;
pmess   equ     9h      ;print message function
display equ     2h      ;display output char fn
doscall equ     21h     ;DOS interrupt address
;
;***********************************************
;
;segment to define ES
;
x_seg   segment
x_byte  db      ?
x_seg   ends
;***********************************************
;
;variable storage area
;
var_area segment
;
lines    db     ?
div_line db     ?
```

```
gp_div   db     ?
header   db     '        0000     2000     4000     6000
                 8000     A000     C000     E000$'
colon    db     ': $'
footer   db     0ah,0dh,'X     X     X     X     X     X     X     X'
         db     0ah,0dh,'000   400   800   C00   1000  1400  1800  1C00$'
;
var_area ends
;*****************************************
;
memscan segment
;
;-----------------------------------------
main     proc    far
;
         assume cs:memscan, ds:var_area
         assume es:x_seg
;
;set up stack for return
start:
         push    ds       ;save DS
         sub     ax,ax    ;clear AX to 0
         push    ax       ;push 0
;
;
;set data segment to work area
         mov     ax,var_area
         mov     ds,ax
;
;initialization
         mov lines,16d    ;lines per screen = 16
         sub     ax,ax    ;set AX to 0
         mov     es,ax    ;set ES register to 0
;
;print header
         call crlf        ;skip a line
         mov  dx,offset header ;addr of message
         mov  ah,pmess    ;print mess function
         int  doscall     ;call DOS
         call crlf        ;skip a line
;
;start a new line
new_line:
;print contents of ES register at start of line
         mov     bx,es    ;print contents of ES
         call    binihex ;  on screen in hex
;print colon and space
         mov  dx,offset colon
```

```
        mov  ah,pmess   ;print message funct
        int doscall     ;call DOS
;
        mov div_line,8d ;divisions per line = 8
        mov     bx,Ø    ;set BX to zero
;
;start a new division
new_div:
        mov gp_div,8d   ;groups/division = 8
;
;start a new group
new_grp:
        mov cx,128d     ;samples per group = 128d
;
;read two bytes and compare them
new_byte:
        mov     al,[bx + x_byte]
        mov     ah,[bx + x_byte + 2]
        and  ax,7f7fh   ;mask off high bits
        cmp     al,ah   ;does 1st = 2nd ?
        je      nx_byte ;yes, nothing here
;
;may be something, look at third byte
        mov     dl,[bx + x_byte + 5]
        and     dl,7fh  ;mask off high bit
        cmp     al,dl   ;does 1st = 3rd ?
        je      nx_byte ;yes
        cmp     ah,dl   ;does 2nd = 3rd ?
        je      nx_byte ;yes
;
;three bytes all different found,
;  so there's something in this group
        mov     dl,'X'  ;print "X"
        mov ah,display  ;display function
        int     doscall ;call DOS
;
;advance BX to next group
        and  bx,ØfcØØh  ;mask off lower 1Ø bits
        add  bx,1Ø24d   ;add 1Ø24d (4ØØh)
        jmp  done_grp   ;done this group
;
;nothing found yet, get next byte
nx_byte:
        add     bx,8d   ;increment byte pointer
        dec     cx      ;done this group?
        jnz   new_byte  ;not yet
;done group, so print a period
        mov     dl,'.'  ;char in DL
```

```
        mov ah,display  ;display function
        int     doscall ;call DOS
;
;we've done one group (1Ø24 bytes)
done_grp:
        dec  gp_div     ;done a division?
        jnz  new_grp    ;no, do next group
        mov  dl,' '     ;yes, print space
        mov  ah,display ;display function
        int  doscall    ;call DOS
;
;we've done one division (8 groups)
done_div:
        dec  div_line   ;done 8 divisions?
        jnz  new_div    ;no, do next division
        call crlf       ;yes, print cr & lf
        mov  ax,es      ;advance the ES
        add  ax,1ØØØh.  ;  to next segment
        mov  es,ax      ;  (add 65536d)
;
;we've done one line (8 segments)
done_line:
        dec  lines      ;done 16 lines?
        jnz  new_line   ;no, do next line
;
;print out values of X positions on bottom row
        mov  dx,offset footer
        mov  ah,pmess   ;print message function
        int  doscall    ;call DOS
        ret             ;yes, return to DOS
;
main    endp
;-----------------------------------------
;
crlf    proc    near

        mov     dl,Ødh  ;carriage return
        mov  ah,display ;display function
        int  doscall    ;call DOS
        mov     dl,Øah  ;linefeed
        mov  ah,display ;display function
        int  doscall    ;call DOS
        ret
crlf    endp
;-----------------------------------------
;
binihex proc    near
;
;subroutine to convert binary number in BX
```

```
;  to hex and print out on console screen
;
        mov     ch,4    ;number of digits
rotate: mov     cl,4    ;set count to 4 bits
        rol     bx,cl   ;left digit to right
        mov     al,bl   ;move digit to AL
        and     al,Øfh  ;mask off left digit
        add     al,3Øh  ;convert hex to ASCII
        cmp     al,3ah  ;is it > 9 ?
        jl      printit ;no, so Ø to 9 digit
        add     al,7h   ;yes, so A to F digit
printit:
        mov     dl,al   ;put ASCII char in DL
        mov ah,display  ;display function
        int  doscall    ;call DOS
        dec     ch      ;done 4 digits?
        jnz     rotate  ;not yet, do another
        ret             ;done subroutine
;
binihex endp
;-----------------------------------------
;
memscan ends
;*****************************************
        end     start
```

HEXIDEC

HEXIDEC is a hexadecimal to decimal converter program. You type a positive hex number from 0 to FFFF on the keyboard, and the program will print out the decimal equivalent, from 0 to 65535, on the screen. To exit from the program, type Ctrl Break. This program is the reverse of the DECIHEX program developed in chapter 6.

Here's an example of output from the program:

```
A>hexidec
Ø           <— Enter a hex number
ØØØØØ       <— Program prints out decimal equivalent
1Ø          <— Hex
ØØØ16       <— Decimal
fff
Ø4Ø95
1ØØØ
Ø4Ø96
8ØØØ
```

```
32768
ffff
65535
^C          ← Type Ctrl Break to exit
A>
```

Here's the listing for HEXIDEC:

```
;HEXIDEC--Main program
;  Converts hex on keyboard to dec on screen
;
hexidec segment
;-----------------------------------------
;
main    proc    far
;
        assume  cs:hexidec
;
;MAIN PART OF PROGRAM.  Links subroutines
;  together.
;
display equ     2h      ;video output
key_in  equ     1h      ;keyboard input
doscall equ     21h     ;DOS interrupt number
;
        push    ds      ;ds on stack
        sub     ax,ax   ;set ax=Ø
        push    ax      ;zero on stack
;
        call    hexibin ;keyboard to binary
        call    crlf    ;print cr & linefeed
;
        call    binidec ;binary to decimal
        call    crlf    ;print cr & linefeed
        jmp     main    ;get next input
;
main    endp
;-----------------------------------------
;
hexibin proc    near
;
;SUBROUTINE TO CONVERT HEX ON KEYBD TO BINARY
;  result is left in BX register
;
        mov     bx,Ø    ;clear BX for number
;
;get digit from keyboard, convert to binary
newchar:
```

```
        mov   ah,key_in ;keyboard input
        int     doscall ;call DOS
        sub     al,3Øh  ;ASCII to binary
        jl      exit    ;jump if < Ø
        cmp     al,1Ød  ;is it > 9d ?
        jl      add_to  ;yes, so it's digit
;
;not digit (Ø to 9), maybe letter (A to F)
        sub     al,27h  ;convert ASCII to bin
        cmp     al,Øah  ;is it < Øa hex?
        jl      exit    ;yes, not letter
        cmp     al,1Øh  ;is it > Øf hex?
        jge     exit    ;yes, not letter
;
;is hex digit. Add to number in BX
add_to:
        mov     cl,4    ;set shift count
        shl     bx,cl   ;rotate BX 4 bits
        mov     ah,Ø    ;zero out AH
        add     bx,ax   ;add digit to number
        jmp     newchar ;get next digit
exit:
        ret
;
hexibin endp
;-----------------------------------------
;
binidec proc    near
;
;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
;  TO DECIMAL ON CONSOLE SCREEN
;
        mov     cx,1ØØØØd ;divide by 1ØØØØ
        call    dec_div
        mov     cx,1ØØØd  ;divide by 1ØØØ
        call    dec_div
        mov     cx,1ØØd ;divide by 1ØØ
        call    dec_div
        mov     cx,1Ød  ;divide by 1Ø
        call    dec_div
        mov     cx,1d   ;divide by 1
        call    dec_div
        ret             ;return from binidec
;
;- - - - - - - - - - - - - - - - - - - - - -
;
dec_div proc    near
;
;sub-subroutine to divide number in BX by
```

```
;  number in CX, print quotient on screen
;  (numerator in AX+DX, denom in CX)
;
        mov     ax,bx   ;number high half
        mov     dx,Ø    ;zero out low half
        div     cx      ;divide by CX
        mov     bx,dx   ;remainder into BX
        mov     dl,al   ;quotient into DL
;
;print the contents of DL on screen
        add     dl,3Øh  ;convert to ASCII
        mov ah,display  ;display function
        int doscall     ;call DOS
        ret             ;return from dec_div
;
dec_div endp
;- - - - - - - - - - - - - - - - - - - - - - -
;
binidec endp
;-----------------------------------------
;
crlf    proc    near
;
;prints carriage return and linefeed
;
        mov     dl,Øah  ;linefeed
        mov  ah,display ;display function
        int     doscall ;call DOS
;
        mov     dl,Ødh  ;carriage return
        mov  ah,display ;display function
        int     doscall ;call DOS
        ret
;
crlf    endp
;-----------------------------------------
;
hexidec ends
        end     main
;
```

PRIME

Two articles in *Byte* magazine (September 1981 and January 1983) described a program that finds prime numbers, using a method called the "sieve of Eratosthenes." Besides being an interesting method for finding primes, the program is also widely used as an informal

benchmark for testing the speed of different computer languages. Programs in a wide variety of languages, running on different computers, have been written to find primes up to 16381.

In the sieve of Eratosthenes method, all the integers (1, 2, 3, etc.) up to a certain value are thought of as being placed on a list. First all the multiples of two are crossed out, leaving only odd numbers. Then all the multiples of three are crossed out, then the multiples of four, and so on. When this process is completed, only prime numbers are left on the list, since those that are a multiple of some other number have been crossed out.

The *Byte* articles reported an enormous variation in the speed of different languages on different computers. (Ten iterations were used to avoid excessively short times for the fastest computers.) Large mainframes running assembly language were, as might be expected, the fastest: the IBM 3033 took 0.0078 seconds. Many of the slower BASICs and other higher-level languages, running on smaller computers, took up to 3000 seconds.

We thought it would be interesting to write versions of this program in BASIC and assembly language on the IBM PC, and compare their speeds of operation. We won't attempt to analyze in detail how the programs work, but you might find analyzing them to be an interesting exercise.

Here's the BASIC version of the program:

```
10 DEFINT A-Z
20 DIM MARK(16381)
25 COUNT=0
30 FOR I=3 TO 16381 STEP 2
40    MARK(I)=0
50 NEXT I
60 FOR I=3 TO 16381 STEP 2
70    IF MARK(I)=1 THEN GOTO 130
80    'PRINT I;
85    COUNT=COUNT+1
90    J=I
100   FOR J=J+I TO 16381 STEP I
110      MARK(J)=1
120   NEXT J
130 NEXT I
135 PRINT : PRINT COUNT "primes"
140 STOP
```

This BASIC program took 3 minutes and 3 seconds, or 1830 seconds to complete the ten iterations of the process.

Here's the assembly language version of the program:

```
;PRIMES3--Finds all prime numbers to 16381
;          using sieve of Eratosthenes
;          prints out total number

max     equ     16381d  ;highest number
display equ     2h      ;display char function
print_m equ     9h      ;print message function
doscall equ     21h     ;DOS interrupt number

;*********************************************

datarea segment ;define data segment

numbers db  max dup (?) ;buffer for integers
mess1   db  'calculating. . .$'
mess2   db  '   $'      ;three spaces

datarea ends
;*********************************************

pro_nam segment         ;define code segment

;---------------------------------------------

main    proc    far     ;main part of program

        assume  cs:pro_nam,ds:datarea

start:          ;starting execution address

;set up stack for return
        push    ds      ;save DS on stack
        sub     ax,ax   ;set AX to zero
        push    ax      ;put on stack

;set DS to data segment
        mov  ax,datarea
        mov  ds,ax

;PRINT INITIAL MESSAGE
        mov  dx,offset mess1  ;load address
        mov  ah,print_m ;print message function
        int  doscall    ;call DOS

;SET COUNT OF PRIMES TO ZERO (CX REGISTER)
        mov  cx,Ø       ;set CX to Ø
```

```
;FILL ODD NUMBERS IN ARRAY WITH ZERO

        mov  bx,3       ;start at 3
fill_Ø:
        mov  [numbers + bx],Ø  ;insert Ø
        inc  bx         ;skip to next
        inc  bx         ;  odd number
        cmp  bx,max     ;done yet?
        jle  fill_Ø     ;not yet

;FIND PRIMES, CROSS OUT ALL THE NONPRIMES

        mov  bx,3       ;start with 3

;has this number been flagged as a nonprime?
search:
        mov  al,[numbers + bx]  ;number into AL
        cmp  al,1       ;is it marked with 1 ?
        je   go_next    ;yes, so it's nonprime

;no, so it's a prime.

;the semicolon can be removed from the
;following line to print out the primes
;       call  binidec   ;print the prime
        inc   cx        ;count the prime

;cross out all the numbers that are multiples
;   of this prime, by marking them "1"
        mov  si,bx      ;j=i
cross_out:
        add  si,bx      ;j=j+i
        cmp  si,max     ;gone too far yet?
        jg   go_next    ;yes
        mov  [numbers + si],1  ;cross it out
        jmp  cross_out  ;do next one

;have we looked at all the numbers?
go_next:
        inc  bx         ;skip to next
        inc  bx         ;  odd number
        cmp  bx,max     ;are we done?
        jle  search     ;not yet

;PRINT OUT TOTAL NUMBER OF PRIMES AND RETURN
        mov  bx,cx      ;put count in BX
        call binidec    ;print it

        ret     ;return from program to DOS
```

```
main    endp    ;end of main part of program
;-----------------------------------------
binidec proc    near

;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
;  TO DECIMAL ON CONSOLE SCREEN

        push    bx      ;save BX
        push    cx      ;save CX

;print three spaces
        mov     dx,offset mess2 ;spaces message
        mov     ah,print_m ;print message func
        int     doscall    ;call DOS

;divide by successive powers of 10d
        mov     cx,10000d ;divide by 10000
        call    dec_div
        mov     cx,1000d  ;divide by 1000
        call    dec_div
        mov     cx,100d ;divide by 100
        call    dec_div
        mov     cx,10d  ;divide by 10
        call    dec_div
        mov     cx,1d   ;divide by 1
        call    dec_div

        pop     cx      ;restore CX
        pop     bx      ;restore BX

        ret             ;return from binidec
;- - - - - - - - - - - - - - - - - - - - - -

dec_div proc    near

;sub-subroutine to divide number in BX by
;  number in CX, print quotient on screen

        mov     ax,bx   ;put number in AX
        cwd     ;ax into ax and dx
        div     cx      ;divide by CX
        mov     bx,dx   ;remainder into BX
        mov     dl,al   ;quotient into DL

;print the contents of DL on screen
        add     dl,30h  ;convert to ASCII
        mov ah,display  ;display function
        int doscall     ;call DOS
        ret             ;return from dec_div
```

```
dec_div endp
;- - - - - - - - - - - - - - - - - - - - - - -

binidec endp
;-----------------------------------------
pro_nam ends    ;end of code segment
;*****************************************

        end     start   ;end assembly
```

The assembly-language program took 8 seconds for ten iterations, which is about 230 times faster than the BASIC program. If there's a more compelling argument favoring assembly language over higher-level languages, especially interpreted ones, we don't know what it is.

There is a CALL BINIDEC instruction in the middle of the PRIME program, preceded by a semicolon so that it becomes a comment. If the semicolon is removed, the prime numbers will be printed out as the program runs. (Of course the printing process slows down the program, so you can't leave this instruction in when you're measuring the speed of the program.) With the semicolon removed, the following printout of prime numbers is generated (we haven't shown all of it).

```
A>prime
calculating. . . ØØØØ3   ØØØØ5   ØØØØ7   ØØØ11   ØØØ13   ØØØ17   ØØØ19   ØØØ23
ØØØ29   ØØØ31   ØØØ37   ØØØ41   ØØØ43   ØØØ47   ØØØ53   ØØØ59   ØØØ61   ØØØ67
ØØØ71   ØØØ73   ØØØ79   ØØØ83   ØØØ89   ØØØ97   ØØ1Ø1   ØØ1Ø3   ØØ1Ø7   ØØ1Ø9
ØØ113   ØØ127   ØØ131   ØØ137   ØØ139   ØØ149   ØØ151   ØØ157   ØØ163   ØØ167
ØØ173   ØØ179   ØØ181   ØØ191   ØØ193   ØØ197   ØØ199   ØØ211   ØØ223   ØØ227
ØØ229   ØØ233   ØØ239   ØØ241   ØØ251   ØØ257   ØØ263   ØØ269   ØØ271   ØØ277
ØØ281   ØØ283   ØØ293   ØØ3Ø7   ØØ311   ØØ313   ØØ317   ØØ331   ØØ337   ØØ347
```

etc., up to 16381.

The Birthday Programs

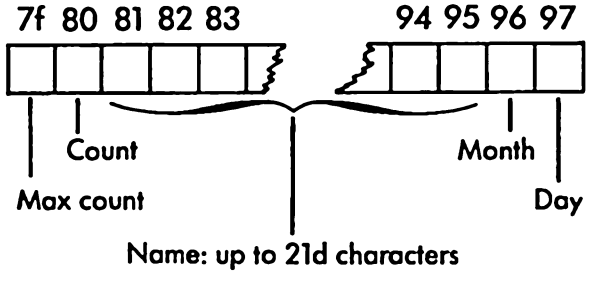
The three programs in this section are used to create, use, and modify a file of birthdays. As we noted in chapter 11, these programs serve as an example of the use of records in disk files, and of the sequential method of disk access.

The first program, SET-BD, creates the birthday file. It prompts you for a person's name, and then for the month and day of the person's birthday. Then it repeats the process for the next person, until it has recorded all the birthdays you wanted it to. This information is stored in a disk file, with one record used for each person.

On any given day you can then call up the program GET-BD from DOS. This program reads the system clock to find out the date, then reads the birthday file to see if anyone on the list has a birthday with that date. If someone does, the name is printed out, and you can then call to wish your friend a happy birthday.

The third program, MOD-BD, is used to modify the birthday file, or to add additional names to it (SET-BD can only be used once).

The files used by these programs consist of records which contain 24d bytes each. The first byte is a count of the number of letters in the name. The name follows, with up to 21d characters. The last two bytes of the record contain the numbers representing the month and the day of the person's birthday. This buffer is located in the default DMA area of the FCB, from 80h to 97h, as shown in the figure below:



Of course the ideas used in these birthday programs can be extended to other programs which create much more complex records. Instead of simply a birthday, records can contain account numbers, payment dates, balances, and so on. This is how files are created for use in mailing lists, accounts receivable, database, and other similar programs.

The SET-BD Program

As noted above, SET-BD is used to create the file of birthdays. When you call it from DOS you must also enter the name of the file the birthdays will be written to. If you forget to do this, the program will simply return to DOS.

Here's an example of the use of the SET-BD program:

```
A>set-bd test.bd                    ← You must enter the name of the file
Enter name: George Jones            )
Enter month of birth (1-12): 5      } —— First person
Enter day of birth (1-31): 15       )
Enter name: Emily Emlen             )
Enter month of birth (1-12): 6      } ——Second person
Enter day of birth (1-31): 16       )
Enter name: Carole Christian        )
Enter month of birth (1-12): 9      } ——Third person
Enter day of birth (1-31): 17       )
Enter name:                         ← Press ↵ to terminate program
```

In this case we've only entered three names; you can enter as many as you like.

Here's the listing for the SET-BD program:

```
;SET-BD--Program to write name and birthday
;          to file

;name of file must follow "set-bd",
;          as in  A>set-bd birthday.txt

;asks for name, and day and month of birthday
;writes this information as one record of file
;   record is 22d bytes long
;repeat as often as necessary
;terminate with enter for name
;NOTE: program can only be used ONCE PER FILE
;uses Sequential Write DOS function

doscall equ  21h        ;DOS interrupt number
create  equ  16h        ;Create File function
w_seq   equ  15h        ;Write Sequential rec
close_f equ  1Øh        ;Close File function
print_m equ   9h        ;print message function
buff_in equ  Øah        ;buffered keybd input
key_in  equ   1h        ;keybd input (1 char)
display equ   2h        ;display output funct
name_sz equ  21d        ;max length of name
rec_sz  equ  name_sz+3  ;name plus 3 bytes
fcb     equ  5ch

;*******************************************
datarea segment         ;define data segment

;DEFINE FILE CONTROL BLOCK
```

```
        org   6ah       ;record size field
r_field db    ?
        org   7ch       ;record number
recno   db    ?


;DEFINE DATA TRANSFER AREA

        org   7fh       ;input buffer
buffer  label byte
max_cnt db    name_sz   ;length of name

        org   80h       ;data transfer area
count   db    ?         ;filled in by buff_in
bname   db    name_sz dup (?) ;buffer for name
month   db    ?         ;byte for month
day     db    ?         ;byte for day

;this data is in segment set by LINK to DATAREA

mess1   db    'Enter name: $'
mess2   db    'Enter month of birth (1-12): $'
mess3   db    'Enter day of birth (1-31): $'

datarea ends
;*********************************************

pro_nam segment         ;define code segment

;---------------------------------------------
main    proc    far     ;main program

        assume cs:pro_nam,ds:datarea

;SET UP STACK FOR RETURN
start:
        push ds         ;save DS
        sub  ax,ax      ;set AX to zero
        push ax         ;put zero on stack

;CREATE FILE

        mov  dx,fcb     ;FCB address in DX
        mov  ah,create  ;Create File function
        int  doscall    ;call DOS
        or   al,al      ;see if found (AL=0)?
        jnz  done       ;no room in directory

;SET RECORD SIZE, RECORD NUMBER
```

```
        mov  r_field,rec_sz ;set size of record
        mov  recno,Ø    ;set record count to Ø


;GET INFORMATION FROM USER
next_rec:

;print "name" message
        mov  dx,offset mess1  ;addr in DX
        call p_mess     ;print message

;set maximum  name size in input buffer
        mov  max_cnt,name_sz

;get name
        mov  dx,offset buffer ;set addr of buff
        mov  ah,buff_in ;keyboard input funct
        int  doscall    ;call DOS
        call crlf       ;new line

;check if name has no characters (user is done)
        mov  al,count   ;get # of chars input
        or   al,al      ;is it zero?
        jz   done       ;yes, user is done

;print "month" message
        mov  dx,offset mess2  ;addr in DX
        call p_mess     ;print message

;get month
        call decibin    ;get month
        mov  month,bl   ;put month in buffer
        call crlf       ;new line

;print "day" message
        mov  dx,offset mess3  ;addr in DX
        call p_mess     ;print message

;get day
        call decibin    ;get day
        mov  day,bl     ;put day in buffer
        call crlf       ;new line

;WRITE RECORD FROM BUFFER TO DISK

        mov  dx,fcb     ;FCB address in DX
        mov  ah,w_seq   ;write sequential func
        int  doscall    ;call DOS
        jmp  next_rec   ;go get next record
```

```
;DONE. SO CLOSE FILE. RETURN TO DOS

done:
        mov  dx,fcb     ;FCB address in DX
        mov  ah,close_f ;Close File function
        int  doscall    ;call DOS
        ret             ;return to DOS

main    endp            ;end of main program
;-----------------------------------------
decibin proc    near

;SUBROUTINE TO CONVERT DEC ON KEYBD TO BINARY
;   result is left in BX register

        mov     bx,0    ;clear BX for number

;get digit from keyboard, convert to binary
newchar:
        mov   ah,key_in ;keyboard input
        int     doscall ;call DOS
        sub     al,30h  ;ASCII to binary
        jl      exit    ;jump if < 0
        cmp     al,9d   ;is it > 9d ?
        jg      exit    ;yes, not dec digit
        cbw     ;byte in AL to word in AX
;(digit is in AX)
;
;multiply number in bx by 10 decimal
        xchg    ax,bx   ;trade digit & number
        mov     cx,10d  ;put 10 dec in CX
        mul     cx      ;number times 10
        xchg    ax,bx   ;trade number & digit
;
;add digit in ax to number in bx
        add     bx,ax   ;add digit to number
        jmp     newchar ;get next digit
exit:
        ret
;
decibin endp
;-----------------------------------------

p_mess  proc    near

;SUBROUTINE TO PRINT MESSAGES
;  enter w address of message in dx
```

```
        push ds          ;save old value of DS
        mov  ax,datarea  ;put new data seg
        mov  ds,ax       ;    in DS
        mov  ah,print_m  ;print message function
        int  doscall     ;call DOS
        pop  ds          ;restore old DS value
        ret

p_mess  endp
;-----------------------------------------

crlf    proc    near

;CRLF--Subroutine to print carriage return
;       and linefeed

        mov     dl,0ah   ;linefeed
        mov  ah,display  ;display function
        int     doscall  ;call DOS

        mov     dl,0dh   ;carriage return
        mov  ah,display  ;display function
        int     doscall  ;call DOS
        ret

crlf    endp
;-----------------------------------------

pro_nam ends             ;end of code segment
;*****************************************
        end     start    ;end of assembly
```

The GET-BD Program

When GET-BD is first called it must be followed by the name of the file containing the birthdays (just as SET-BD was). It will then read the system clock to determine the date, and check the file for matches with this date. If it finds any, it will print out the person's name. In the example below we use the DATE function just so you can see it really is September 17th; you wouldn't normally need to do this.

```
A>date                          <— Call DOS DATE function
Current date is Sat  9-17-1983  <— It's September 17th
Enter new date:                 <— Press (↵)

A>get-bd test.bd                <— Call GET-BD
Carole Christian                <— Phone Carole, tell her "Happy Birthday!"
```

Here's the listing of the GET-BD program.

```
;GET-BD--Program to search file for names
;        with today's birthday

;name of file must follow "get-ed".  E.g.:
;          A>get-bd birthday.txt
;if no matches, nothing will be printed

;uses sequential read

doscall equ  21h        ;DOS interrupt number
open_f  equ  Øfh        ;Open File function
r_seq   equ  14h        ;Read Sequential rec
display equ   2h        ;display output funct
g_date  equ  2ah        ;get date function
name_sz equ  21d        ;max length of name
rec_sz  equ  name_sz+3  ;name plus 3 bytes
fcb     equ  5ch

;*********************************************
datarea segment         ;define data segment

;DEFINE FILE CONTROL BLOCK

        org   6ah       ;record size field
r_field db    ?
        org   7ch       ;record number
recno   db    ?


;DEFINE DATA TRANSFER AREA

        org   7fh       ;input buffer
max_cnt db    name_sz   ;max length of name

        org   8Øh       ;data transfer area
count   db    ?         ;filled in by buff_in
bname   db    name_sz dup (?) ;buffer for name
month   db    ?         ;byte for month
day     db    ?         ;byte for day


datarea ends
;*********************************************

pro_nam segment         ;define code segment
```

```
;-----------------------------------------
main    proc    far     ;main program

        assume cs:pro_nam,ds:datarea

;SET UP STACK FOR RETURN
start:
        push ds         ;save DS
        sub  ax,ax      ;set AX to zero
        push ax         ;put zero on stack

;OPEN FILE

        mov  dx,fcb     ;FCB address in DX
        mov  ah,open_f  ;Open File function
        int  doscall    ;call DOS
        or   al,al      ;see if found (AL=Ø)?
        jnz  done       ;can't find file

;SET RECORD SIZE, RECORD NUMBER

        mov  r_field,rec_sz ;set size of record
        mov  recno,Ø    ;set record count to Ø

;READ RECORD FROM DISK INTO BUFFER

next_rec:
        mov  dx,fcb     ;FCB address in DX
        mov  ah,r_seq   ;Read Sequential rec
        int  doscall    ;call DOS
        or   al,al      ;end-of-file? (AL< >Ø?)
        jnz  done       ;yes, so we're done

;GET MONTH AND DAY FROM INTERNAL CLOCK

        mov  ah,g_date  ;get date function
        int  doscall    ;call DOS

;COMPARE MONTH AND DAY IN RECORD WITH
;        CURRENT MONTH AND DAY
        cmp  month,dh   ;DH is current month
        jnz  next_rec   ;no match; get next rec
        cmp  day,dl     ;DL is current day
        jnz  next_rec   ;no match; get next rec

;MONTH AND DAY MATCH, SO PRINT NAME
```

```
        mov  cl,count   ;letter count in CX
        mov  ch,Ø       ; (zero out top half)
        mov  bx,offset bname ;start of buffer
new_char:
        mov  dl,[bx]    ;get char from buffer
        mov  ah,display ;display char function
        int  doscall    ;call DOS
        inc  bx         ;bump pointer
        loop new_char   ;do until CX=Ø

        call crlf       ;new line
        jmp  next_rec   ;get next record

;DONE, SO RETURN TO DOS

done:
        ret             ;return to DOS

main    endp            ;end of main program
;-----------------------------------------
crlf    proc    near

;CRLF--Subroutine to print carriage return
;       and linefeed

        mov     dl,Øah  ;linefeed
        mov  ah,display ;display function
        int     doscall ;call DOS

        mov     dl,Ødh  ;carriage return
        mov  ah,display ;display function
        int     doscall ;call DOS
        ret

crlf    endp
;-----------------------------------------

pro_nam ends            ;end of code segment
;*********************************************
        end     start   ;end of assembly
```

The MOD-BD Program

This program is used to modify or add to the file created with SET-BD. It first goes through the names on the list one by one, asking if you want to modify them. If you don't want to modify a particular record, you type "n". At the end of the list it asks if you want to append another

name to the list. If you type "n" the program exits. If you want to modify an existing record, or add another record to the file, just type "y" and fill in the information as in SET-BD.

Here's an example of MOD-BD in operation on the file created above.

```
A>mod-bd test.bd

George Jones
ØØØØ5
ØØØ15
MODIFY THIS RECORD? (y/n): n
Emily Emlen
ØØØØ6
ØØØ16
MODIFY THIS RECORD? (y/n): n
Carole Christian
ØØØØ9
ØØØ17
MODIFY THIS RECORD? (y/n): n
ADD ANOTHER RECORD? (y/n): y
Enter name: Dennis Douglas
Enter month of birth (1-12): 9
Enter day of birth (1-31): 17
```

Now if we run GET-BD again, we'll see two names, since both these people have a birthday on September 17.

```
A>get-bd test.bd
Carole Christian
Dennis Douglas
```

Here's the listing for MOD-BD.

```
;MOD-BD--Program to modify or add to info
;        in birthday file

;name of file must follow "mod-bd".  E.g.:
;          A>mod-bd birthday.txt

;file must already have been created by set-bd

;displays contents of each record, asks if user
;    wants to change it

;uses sequential read and write
```

```
doscall equ  21h        ;DOS interrupt number
open_f  equ  0fh        ;open file function
w_seq   equ  15h        ;write sequential rec
r_seq   equ  14h        ;read sequential rec
close_f equ  10h        ;close file function
print_m equ   9h        ;print message function
buff_in equ  0ah        ;buffered keybd input
key_in  equ   1h        ;keybd input (1 char)
display equ   2h        ;display output funct
name_sz equ  21d        ;max length of name
rec_sz  equ  name_sz+3  ;name plus 3 bytes
fcb     equ  5ch        ;File Control Block
let_y   equ  79h        ;letter "y"
;*********************************************
datarea segment         ;define data segment

;DEFINE FILE CONTROL BLOCK

        org   6ah       ;record size field
r_field db    ?
        org   7ch       ;record number
recno   db    ?


;DEFINE DATA TRANSFER AREA

        org   7fh       ;input buffer
buffer  label byte
max_cnt db    name_sz   ;length of name

        org   80h       ;data transfer area
count   db    ?         ;filled in by buff_in
bname   db    name_sz dup (?) ;buffer for name
month   db    ?         ;byte for month
day     db    ?         ;byte for day

;this data is in segment set by LINK to DATAREA

mess0   db    'MODIFY THIS RECORD? (y/n): $'
mess1   db    'Enter name: $'
mess2   db    'Enter month of birth (1-12): $'
mess3   db    'Enter day of birth (1-31): $'
mess4   db    'ADD ANOTHER RECORD? (y/n): $'

datarea ends
;*********************************************

pro_nam segment         ;define code segment
```

```
;-----------------------------------------
main    proc    far     ;main program

        assume cs:pro_nam,ds:datarea

;SET UP STACK FOR RETURN
start:
        push ds         ;save DS
        sub  ax,ax      ;set AX to zero
        push ax         ;put zero on stack

;OPEN FILE

        mov  dx,fcb     ;FCB address in DX
        mov  ah,open_f  ;open file function
        int  doscall    ;call DOS
        or   al,al      ;see if found (AL=Ø)?
        jz   set_size   ;found OK
        jmp  done       ;no such filename
;(note: this jump too long for "jnz done")

;SET RECORD SIZE, RECORD NUMBER

set_size:
        mov  r_field,rec_sz ;set size of record
        mov  recno,Ø    ;set record count to Ø

;READ RECORD
next_rec:
        call crlf       ;print return, linefeed
        mov  dx,fcb     ;FCB address in DX
        mov  ah,r_seq   ;read sequential funct
        int  doscall    ;call DOS
        or   al,al      ;end-of-file? (AL<>Ø?)
        jz   p_rec      ;no
        jmp  end_file   ;yes
;(note: this jump too far for "jnz end_file")

;PRINT CONTENTS OF RECORD FROM BUFFER
p_rec:

;print name
        mov  cl,count   ;letter count in CX
        mov  ch,Ø       ;  (zero out top half)
        mov  bx,offset bname ;start of buffer
new_char:
        mov  dl,[bx]    ;get char from buffer
        mov  ah,display ;display char function
```

```
        int  doscall    ;call DOS
        inc  bx         ;bump pointer
        loop new_char   ;do until CX=Ø
        call crlf       ;new line

;print month
        mov  bl,month   ;month from buffer
        mov  bh,Ø       ; (zero out top half)
        call binidec    ;print in decimal
        call crlf       ;new line

;print day
        mov  bl,day     ;day from buffer
        mov  bh,Ø       ; (zero out top half)
        call binidec    ;print in decimal
        call crlf       ;new line

;ASK IF USER WANTS TO CHANGE IT

;print "modify?" message
        mov  dx,offset messØ ;addr in DX
        call p_mess     ;print message

;get "y" or "n" answer
        mov  ah,key_in  ;keyboard input funct
        int  doscall    ;call DOS
        cmp  al,let_y   ;is it letter "y" ?
        jne  next_rec   ;no, go read next rec
;yes, so user wants to modify record
        dec  recno      ;decrement record #

;GET INFORMATION FROM USER
mod_rec:
        call crlf       ;print return, linefeed

;print "name" message
        mov  dx,offset mess1  ;addr in DX
        call p_mess     ;print message

;set maximum  name size in input buffer
        mov  max_cnt,name_sz

;get name
        mov  dx,offset buffer ;set addr of buff
        mov  ah,buff_in ;keyboard input funct
        int  doscall    ;call DOS
        call crlf       ;new line
```

```
;check if name has no characters (user is done)
        mov  al,count    ;get # of chars input
        or   al,al       ;is it zero?
        jz   next_rec    ;yes, get next record

;print "month" message
        mov  dx,offset mess2  ;addr in DX
        call p_mess      ;print message

;get month
        call decibin     ;get month
        mov  month,bl    ;put month in buffer
        call crlf        ;new line

;print "day" message
        mov  dx,offset mess3  ;addr in DX
        call p_mess      ;print message

;get day
        call decibin     ;get day
        mov  day,bl      ;put day in buffer
        call crlf        ;new line

;WRITE RECORD FROM BUFFER TO DISK

        mov  dx,fcb      ;FCB address in DX
        mov  ah,w_seq    ;write sequential func
        int  doscall     ;call DOS
        jmp  next_rec    ;go get next record

;END OF FILE.  SEE IF USER WANTS TO ADD RECORD
end_file:

;print "add another record" message
        mov  dx,offset mess4 ;addr of message
        call p_mess      ;print message

;get "y" or "n" answer
        mov  ah,key_in   ;keyboard input funct
        int  doscall     ;call DOS
        cmp  al,let_y    ;is it letter "y" ?
        je   mod_rec     ;yes, go add new rec
;no, so user is done.

;DONE, SO CLOSE FILE, RETURN TO DOS
```

```
done:
        mov  dx,fcb     ;FCB address in DX
        mov  ah,close_f ;close file function
        int  doscall    ;call DOS
        ret             ;return to DOS

main    endp            ;end of main program

;-----------------------------------------
;
binidec proc    near
;
;SUBROUTINE TO CONVERT BINARY NUMBER IN BX
;  TO DECIMAL ON CONSOLE SCREEN
;
        mov     cx,10000d ;divide by 10000
        call    dec_div
        mov     cx,1000d  ;divide by 1000
        call    dec_div
        mov     cx,100d ;divide by 100
        call    dec_div
        mov     cx,10d  ;divide by 10
        call    dec_div
        mov     cx,1d   ;divide by 1
        call    dec_div
        ret             ;return from binidec
;
;- - - - - - - - - - - - - - - - - - - - - -
;
dec_div proc    near
;
;sub-subroutine to divide number in BX by
;  number in CX, print quotient on screen
;
        mov     ax,bx   ;put number in AX
        cwd     ;ax into ax and dx
        div     cx      ;divide by cx
        mov     bx,dx   ;remainder into BX
        mov     dl,al   ;quotient into DL
;
;print the contents of DL on screen
        add     dl,30h  ;convert to ASCII
        mov ah,display  ;display function
        int doscall     ;call DOS
        ret             ;return from dec_div
```

```
;
dec_div endp
;- - - - - - - - - - - - - - - - - - - - - - - -
;
binidec endp
;-----------------------------------------



;-----------------------------------------
decibin proc    near

;SUBROUTINE TO CONVERT DEC ON KEYBD TO BINARY
;   result is left in BX register

        mov     bx,Ø    ;clear BX for number

;get digit from keyboard, convert to binary
newchar:
        mov   ah,key_in ;keyboard input
        int     doscall ;call DOS
        sub     al,3Øh  ;ASCII to binary
        jl      exit    ;jump if < Ø
        cmp     al,9d   ;is it > 9d ?
        jg      exit    ;yes, not dec digit
        cbw     ;byte in AL to word in AX
;(digit is in AX)
;
;multiply number in bx by 1Ø decimal
        xchg    ax,bx   ;trade digit & number
        mov     cx,1Ød  ;put 1Ø dec in CX
        mul     cx      ;number times 1Ø
        xchg    ax,bx   ;trade number & digit
;
;add digit in ax to number in bx
        add     bx,ax   ;add digit to number
        jmp     newchar ;get next digit
exit:
        ret
;
decibin endp
;-----------------------------------------

p_mess  proc    near

;SUBROUTINE TO PRINT MESSAGES
;  enter w address of message in dx
```

```
        push ds          ;save old value of DS
        mov  ax,datarea  ;put new data seg
        mov  ds,ax       ;    in DS
        mov  ah,print_m  ;print message function
        int  doscall     ;call DOS
        pop  ds          ;restore old DS value
        ret

p_mess  endp
;-----------------------------------------

crlf    proc    near

;CRLF--Subroutine to print carriage return
;      and linefeed

        mov     dl,Øah   ;linefeed
        mov  ah,display  ;display function
        int     doscall  ;call DOS

        mov     dl,Ødh   ;carriage return
        mov  ah,display  ;display function
        int     doscall  ;call DOS
        ret

crlf    endp
;-----------------------------------------

pro_nam ends             ;end of code segment
;*****************************************
        end     start    ;end of assembly
```

SAVEIMAG

SAVEIMAG is a way of saving the screen image as a disk file. You can cause the screen image to be printed on the printer by pressing ⇧/Prtsc; SAVEIMAG performs a similar function, but saves the image to a disk file. This is useful if you want to record in a text file a process that took place on the screen, including prompts displayed by the program as well as input from the keyboard. For instance, all the DEBUG dumps and trace sessions, as shown in the early chapters of this book, were saved to the disk with SAVEIMAG.

SAVEIMAG works by taking all the characters in the monochrome display screen, creating a disk file from them, and writing it to the disk with the Random Block Write DOS function. Attributes are not saved; only the ASCII characters.

This program saves all 80 characters from each line. However, in order to be useful as a text file, the resulting image must have a carriage return inserted at the end of each line. This makes every line 81 characters long. Thus when you use TYPE to look at the contents of a file generated with SAVEIMAG, it will appear to be double-spaced. A word processor program, however, will read it correctly.

Here's the listing of the SAVEIMAG program:

```
;SAVEIMAG--Save image of screen memory on disk
;  in text form ("attributes" not saved)

c_count equ     2000d   ;number of orig chars
new_cnt equ     2050d   ;includes rets & lfs
ch_line equ     80d     ;chars per output line
r_size  equ     80h     ;record size
f_size  equ     new_cnt/r_size ;number of recs

doscall equ     21h     ;DOS interrupt number
create  equ     16h     ;create file function
close_f equ     10h     ;close file function
print_m equ      9h     ;print message funtion
block_w equ     28h     ;rand blk write func
set_dta equ     1ah     ;set Disk Transfer Addr

fcb     equ     5ch     ;File Control Block
screen  equ     0b000h  ;segment of screen mem

eof     equ     1ah     ;end-of-file character
car_ret equ     0dh     ;carriage return char
l_feed  equ     0ah     ;linefeed character

;*********************************************

stacker segment stack   ;define stack segment

        db      40h dup ('stack...')
stacker ends
;*********************************************

datarea segment         ;define data segment

;these items are in the "program segment
;   prefix" segment

        org     6ah
rs_field dw     ?       ;rec size field in FCB
```

```
        org     7dh
r1      dw      ?       ;random rec size (low)
r2      db      ?       ;random rec size (high)

        org     8Øh
buffer  db  new_cnt dup (?)

;these items are in the datarea segment

mess1   db  'No space on disk.$'

datarea ends
;*****************************************

video   segment         ;define extra segment
vid_1   db      (?)
video   ends
;*****************************************

pro_nam segment         ;define code segment

;-----------------------------------------

main    proc    far     ;main part of program

        assume  cs:pro_nam,ds:datarea
        assume  ss:stacker,es:video

start:          ;starting execution address

;set up stack for return
        push    ds      ;save DS
        sub     ax,ax   ;set AX to Ø
        push    ax      ;put it on stack

;MOVE TEXT IN SCREEN MEMORY TO BUFFER

;set up screen pointer
        mov  ax,screen  ;screen memory address
        mov  es,ax      ; into ES
        mov  si,Ø       ;Ø in screen pointer

;set up buffer pointer
        mov  di,Ø        ;Ø in buffer pointer

;set up count
        mov  cx,c_count ;chars in screen mem
        mov  dh,ch_line ;set chars/line count
```

```
;transfer the characters from screen to buffer
transfer:
        mov  al,[vid_1 + si] ;char from screen
        inc  si              ;bump screen pointer
        inc  si              ;  twice
        mov  [buffer + di],al ;put char in buff
        inc  di              ;bump buff pointer once

;if at end of line (multiple of 80d chars),
;  then insert return and linefeed in buffer

        dec  dh              ;done this line?
        jnz  no_return       ;not yet, do next char

        mov  [buffer + di],car_ret ;insert ret
        inc  di              ;bump pointer
        mov  [buffer + di],l_feed  ;insert lf
        inc  di
        mov  dh,ch_line ;reset chars/line count

no_return:
        loop transfer    ;get next character

;CREATE THE FILE

        mov  dx,fcb      ;put FCB addr in DX
        mov  ah, create ;create file function
        int  doscall     ;call DOS
        inc  al          ;if AL was FF, then
        je   no_space    ;  no space to write

;SET RANDOM RECORD FIELD

        mov  r1,0        ;low word
        mov  r2,0        ;high byte

;WRITE BLOCK TO DISK

        mov  cx,f_size   ;put file size in CX
        mov  dx,fcb      ;put FCB address in DX
        mov  ah,block_w ;block write function
        int  doscall     ;call DOS
        or   al,al       ;check if write o.k.
        jnz  no_space    ;if AL not 0, bad write

;CLOSE FILE
```

```
        mov  dx,fcb     ;put FCB addr in DX
        mov  ah,close_f ;close file function
        int  doscall    ;call DOS

exit:   ret             ;return to DOS

;PRINT OUT MESSAGE

no_space:
        mov  dx, offset mess1  ;get message
        push ds         ;save old value of DS
        mov  ax,datarea ;put seg addr of
        mov  ds,ax      ;  datarea in DS
        mov  ah,print_m ;print message function
        int  doscall    ;call DOS
        pop  ds         ;restore value of DS
        jmp  exit

main    endp    ;end of main part of program
;-----------------------------------------
pro_nam ends    ;end of code segment
;*****************************************

        end     start   ;end assembly
```

Index

Robert W. Lafore

ASSEMBLY LANGUAGE PRIMER FOR THE IBM PC

The Waite Group 1984

Acquisizione: Libro in Djvu con un pesante sfondo viola su tutte le pagine trovato online; rendering su file TIF a 600 dpi;

Post-processing: eliminazione sfondi; cropping del testo; deskewing; dewarping; margini equalizzati; miglioramento della nitidezza del testo; pulizia manuale pagine;

Conversione: da TIF a DJVU e da DJVU a PDF.

Ottimizzato per stampa laser in bianco e nero.

Libro non di mia proprietà – Giugno 2015

by Salviati

COMPUTER • Z5711 • $24.95
CANADA • $29.95

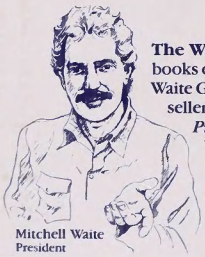
ASSEMBLY LANGUAGE PRIMER for the IBM® PC & XT

Assembly language is the fastest and most powerful language available for any computer. It can access *all* of a machine's features, and it avoids the restrictions of higher-level languages such as Pascal and BASIC. Programs demanding speed or flexibility, such as graphics routines, word processors, and spreadsheets, are best written in assembly language.

Assembly language is often considered difficult to learn, but *Assembly Language Primer* proves this need not be true. Robert Lafore's unique use of DOS functions makes it possible to write programs that perform interesting tasks in just a few easily understood lines. By writing those initial programs with DEBUG, and later with the IBM Assembler ASM, Lafore takes you quickly into assembly language programming, avoiding the complex overhead of the assembler and linker programs.

As you learn assembly language you'll also learn how to use the built-in PC-DOS and ROM functions; how the video memory works; how to write graphics routines; how to make music and sound effects; how to access the keyboard, display, printer, and disk systems; and much more.

Assembly Language Primer for the IBM PC & XT is fully illustrated, contains over 100 example programs, and is completely compatible with other books in the *Plume/Waite* IBM Primer series.



Mitchell Waite
President

The Waite Group is a San Rafael, California based producer of high-quality books on personal computing. Acknowledged as a leader in the industry, the Waite Group has written and produced over thirty titles, including such best sellers as *Unix Primer Plus, Graphics Primer for the IBM PC, CP/M Primer,* and *Soul of CP/M.* Internationally known and award winning, Waite Group books are distributed world-wide, and have been repackaged with the products of such major companies as Epson, Wang, Xerox, Tandy Radio-Shack, NCR and Exxon. Mr. Waite, President of the Waite Group, has been involved in the computer industry since 1972 when he bought his first Apple I computer from Steven Jobs.

ISBN 0-452-25711-5