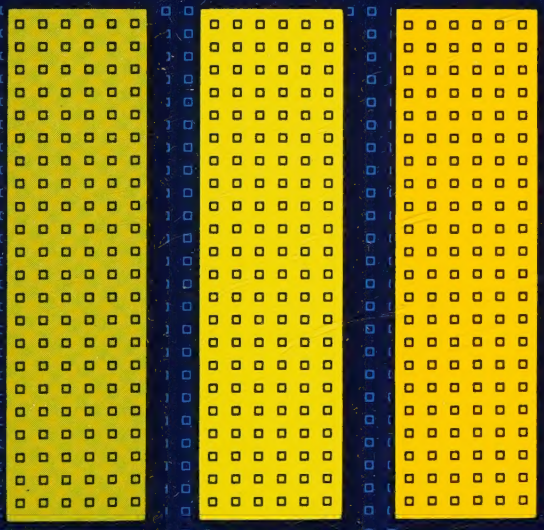
The Waite Group

PASCAL PRIMER for the IBM® PC



Complete Guide to IBM Pascal Programming

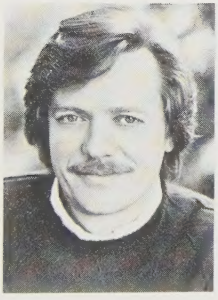
- Covers Novice to Advanced Levels
- For IBM Enhanced Personal Computer Pascal
- Features Graphics and Sound Programming
- Explains Files, Pointers, Lists, and Modules

by Michael Pardee

PASCAL PRIMER for the IBM® PC

This book takes the beginning Pascal programmer on a fascinating educational journey through this important IBM PC language. Pascal is preferred for serious application programs: it is a structured language, which simplifies the writing and management of large programs, and it's compiled, which makes it fast. Pascal's logical syntax also gives it wide acceptance in the educational community.

The extensive programming examples in this book keep reader interest high by using the graphics and sound capabilities of the IBM PC. Experienced computer-book author Michael Pardee takes you easily and quickly from fundamentals to advanced concepts, including file usage, super-arrays, and interfacing Pascal to 8088 assembly language.



Michael Pardee

Michael Pardee is Managing Editor for The Waite Group, a San Rafael, California corporation specializing in the writing and packaging of personal computer books. Mr. Pardee, one of the earliest writers in the personal computer industry, is coauthor, along with Mitchell Waite, of the books *Microcomputer Primer, Your Own Computer,* and the best-selling *BASIC Programming Primer.* Mr. Pardee has extensive experience in business data processing and computerized typesetting for the newspaper industry, and is knowledgeable in several high-level and microprocessor based languages. Mr. Pardee is also an accomplished musician fluent in the guitar, piano and organ. His personal interests include sailing and camping.

PASCAL PRIMER

for the IBM® PC

by

Michael Pardee

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

Pardee, Michael.
Pascal primer for the IBM PC.

"A Plume/Waite book."
Includes index.
1. IBM Personal Computer—Programming. 2. PASCAL (Computer program language) I. Title.
QA76.8.I2594P27 1984 001.64'24 84-2159
ISBN 0-452-25496-5

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

First Printing, May, 1984

1 2 3 4 5 6 7 8 9

Book and cover design by Dan Cooper

Typography by Walker Graphics

PRINTED IN THE UNITED STATES OF AMERICA

Contents

To my Mother

ACKNOWLEDGMENTS

The creation of any book always involves the cooperation of many people. The author would like to take this opportunity to give special thanks to Ken McCreery for his many contributions to this book. Ken provided many of the programming examples for the book, and was closely involved with its overall development. Also to be acknowledged is Robert Lafore for his great editing job, and his patience with the author.

—Michael Pardee

Preface

"more than a tool; it is a tool maker..."

This phrase from the *IBM Personal Computer Pascal* manual describes IBM Pascal perfectly. Because the language is so powerful, it can be used to create almost any kind of program, including such complex programming tools as compilers, assemblers, and operating systems.

Pascal was originally created to teach the art of problem-solving with computers, and it possesses the clear, logical structure that makes learning it such a valuable experience. Every element of a Pascal program is specifically defined, and every statement can be annotated with extensive comments, so a program written in Pascal is much easier to understand than one written in an interpreted language such as BASIC.

Moreover, Pascal has been extended far beyond its academic beginnings. It includes versatile general-purpose input and output capabilities and powerful dynamic memory allocation structures such as "super-arrays." (Don't worry, we'll talk about what this means later.) In addition, IBM has added many enhancements to IBM Personal Computer Pascal, so that while it still supports programs written in standard Pascal, there are many different and exciting things you can do on your IBM that aren't possible with other Pascal implementations. The two-pass compiler optimizes the object code for size and speed, making IBM Pascal an ideal language for business applications, word processing, speech synthesis and recognition, guidance systems, and a vast horizon of other areas. In addition, IBM Pascal is easily "linked" with assembly language modules, so specialized I/O driver routines can be integrated into a Pascal program.

This book is written for anyone who wants to learn IBM Personal Computer Pascal (which we'll call "IBM Pascal" from now on). It will be useful both for the novice, and for the programmer who is already familiar with other higher-level languages such as BASIC. However, the book assumes that the reader has some general knowledge of programming and computers.

Like others in this Plume/Waite series, this book is intended to be your guide to learning about the IBM Personal Computer, and how

Pascal is implemented in this specific environment. It is not just another general-purpose Pascal language manual. There are many examples of Pascal program segments to illustrate the various concepts, all of which are written to take advantage of the IBM's unique capabilities. Also included is a system of routines to provide Pascal with color graphics and sound capabilities.

We hope you enjoy your voyage into the world of Pascal programming, and that you find it fascinating and profitable.

1

The Big Picture

Concepts

Interpreted and compiled languages
Source, Object and Executable files
Required hardware and software
Organizing the diskettes
Sections of a program
Program comments and remarks
Compiling a program
Linking a program
Using DOS Batch files

Keywords

PROGRAM, CONST, TYPE, VAR, VALUE, FUNCTION, PROCEDURE, BEGIN, END

We're about to begin an adventure: learning how to use the Pascal programming language for the IBM Personal Computer. It's going to be an exciting journey. We'll be exploring each of the major areas of Pascal in detail. But first, let's take a look at the big picture. We'll start with some general information about Pascal and the IBM PC. We'll see what hardware and software are necessary, and what's involved in compiling a Pascal program. Then, we'll present a real program. We'll discuss each of the program's major parts to orient you to Pascal programming. Then, in further chapters, when we explore each area in detail, you'll have an idea of how they all fit together.

What Is Pascal?

A few years ago, a new programming language appeared on the computer scene in the United States. Pascal, developed in Switzerland in

1970, had found its way west. Similar in many respects to the language ALGOL, Pascal was developed by Dr. Niklaus Wirth, a professor in Zurich, as a vehicle for teaching a structured approach to problem solving.

The first major implementation of Pascal in the US was at the University of California at San Diego. This version, known as *UCSD Pascal*, has been implemented on several different computers. including the IBM PC. But IBM also has its own version of Pascal. This *IBM Pascal* incorporates all of the standard features of Pascal, and includes many enhancements that are IBM-dependent. This is a book about IBM Pascal, not UCSD Pascal, or any other version.

Oh yes...the name? The Pascal programming language was named after Blaise Pascal, a French mathematician, credited with building the first mechanical computer. It's a great name for a programming language, and connotes a certain degree of class that is definitely in keeping with the nature of the language. Pascal is one of the most beautiful programming languages you'll ever learn. Its logical structures are clean and easy to use. The data types are powerful and all-inclusive. Even the listings are pretty.

Interpreted Versus Compiled

There are basically two types of programming languages in use today. There are *interpreted* languages, like BASICA for the IBM PC, and then there are *compiled* languages, of which Pascal is one. The main difference is this. In an interpreted language like BASICA, the actual program statements are loaded into the computer and the BASIC interpreter performs the operations specified by each statement one at a time. Each statement is interpreted every time the program is executed. By "interpreted" we mean that the BASIC interpreter program figures out what a particular statement means, and converts it into *machine language* instructions — those that can be executed directly by the computer.

In a compiled language, on the other hand, the program statements, called the *source*, are interpreted just once by a program called the *compiler*. The compiler's job is to generate an *object program* that can be linked into an *executable program* of machine language instructions. Then, the compiler's work is finished, at least until changes in the program are necessary.

Advantages of Interpreted Languages

There are advantages and disadvantages of each type of programming language. It's generally quicker to write a program in an

interpreted language. This is mainly due to the fact that the program statements are *resident* in the computer during execution. Therefore, it is possible to:

Enter a program.

Run the program.

Observe the operation.

Make corrections.

Run it again.

In fact, it's even possible to stop a program in mid-execution, make some change and then resume execution at the point where it was stopped.

The major *disadvantage* of an interpreted programming language is that it is slow. This is because each program statement has to be re-interpreted every time before the computer knows what to do. Another disadvantage is that since the source program is right there in memory during execution, the nature of its operation cannot be concealed. This makes it practically impossible to write *proprietary* programs (those you want to sell) with an interpreted language.

Advantages of Compiled Languages

The main advantage of compiled programs is that they execute faster. The program statements have already been compiled into actual machine instructions, so all the computer has to do is execute them. We have found that Pascal executes the same types of operations in about 1/8th the time it takes BASICA to do them.

Another advantage of a compiled language is that its source programs can be generously documented with comments right in line with the code. Since the source program is not loaded into the computer during execution, no consideration need be made for the memory that is "wasted" on the comments. The comments are not compiled. They are only printed on the listing that the compiler produces when it generates the object program.

The major *disadvantage* of a compiled programming language is that it will generally take longer to develop programs and get them running. This is because of the additional steps in compiled program development, as shown in Figure 1-1. First, we have to input the *source statements* via a text editor of some sort. Next, we have to run phase one of the compiler program. If the compiler detects errors in the source program, the compilation will not be completed. Instead, error messages will be inserted in the listing that the compiler produces. The errors must then be corrected, again using the text editor, and the compilation process

repeated. When writing a large, complex program, this process might be repeated many times until the compilation is completed without any errors. The program must then be linked together and run to check for additional errors.

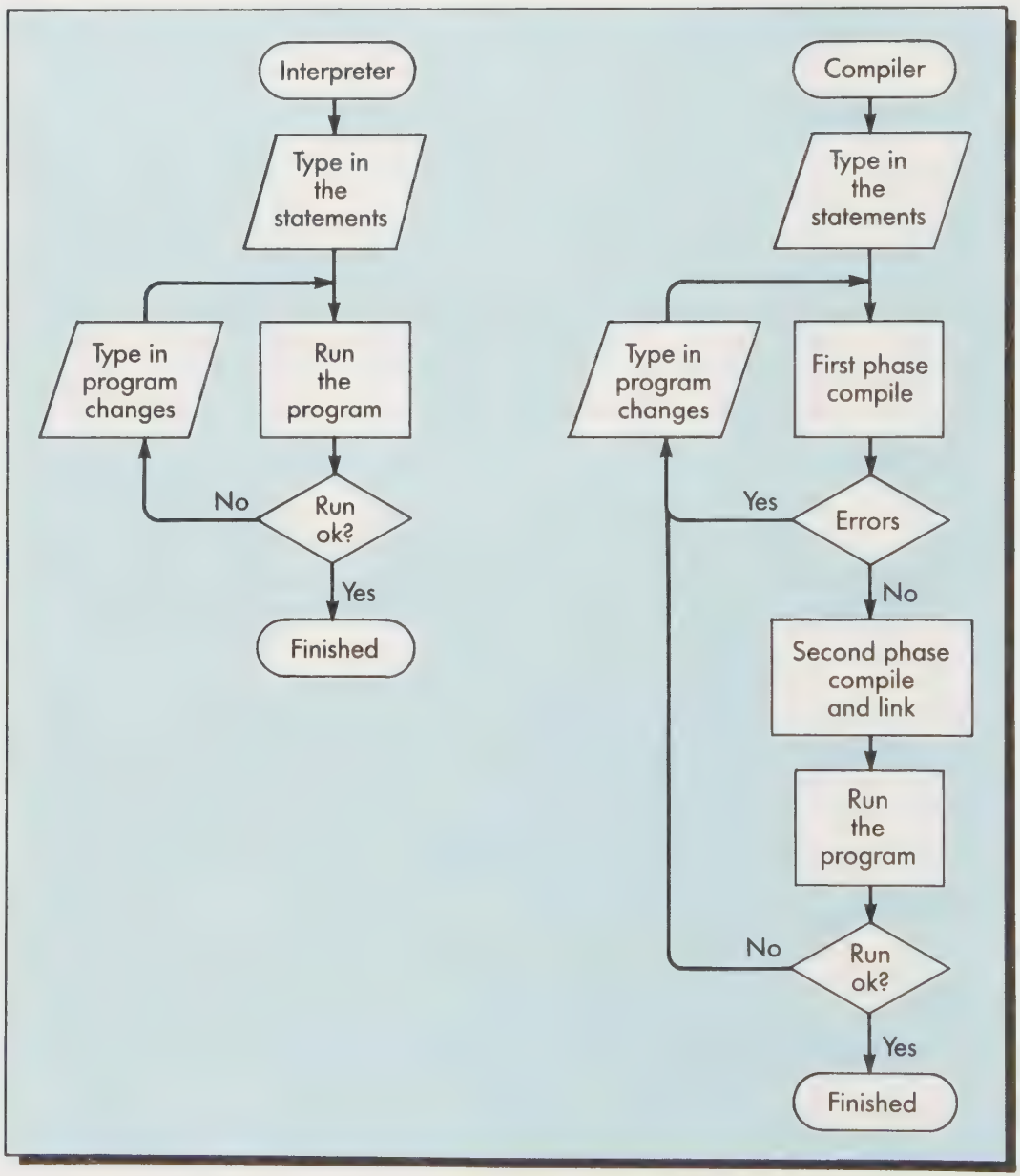


Figure 1-1. Interpreter versus compiler

The choice between compiled and interpreted languages really depends upon the particular programming application. Some applications are better programmed using an interpreter like BASICA. This is especially true for "one time" programs that you just want to get working in a hurry. Other applications will be more suited to a compiled language, especially those which require fast execution speed.

IBM Pascal Versus UCSD Pascal

We mentioned that UCSD Pascal can be used with the IBM PC. In general, the actual Pascal data types and statement structures are similar in IBM Pascal and UCSD Pascal. IBM Pascal and UCSD Pascal support all of the standard Pascal features. But IBM Pascal also provides extensions and enhancements to the standard.

The main difference between the two Pascals has to do with the program development and execution process. UCSD Pascal is a completely self-contained package, called the *p-system*. It has its own text editor, the Pascal compiler; a *runtime* system that oversees the execution of the program, and a special *files* system. It does not use the PC-DOS operating system.

On the other hand, IBM Pascal can use any text editor (EDLIN®, WordStar®, or others). The compiled object program is then converted into an executable form by the *linker* program. This program is stored on the disk with the filename extension EXE and can be run simply by typing the program name at the DOS prompt (A>). IBM Pascal programs use standard IBM files, and access them using PC-DOS routines.

What Hardware Is Necessary?

IBM Pascal is designed to operate on an IBM Personal Computer. The following hardware configuration is necessary to support Pascal.

System Memory of 128K

The system will require a minimum of 128K bytes of read/write memory in order to compile Pascal programs. This is due to the size of the two Pascal compiler phases. The first phase, PAS1, is over 80K bytes; and the second phase, PAS2, is over 99K bytes. This should cause little trouble since most IBM-PCs are sold with at least this much memory.

Dual Diskettes

While it is possible to write a Pascal program with only one disk drive, it becomes quite cumbersome, and is generally not advisable. With two diskettes, the various programs and files necessary for Pascal can be more conveniently allocated, as you will see later in this chapter.

Printer

Although not absolutely necessary for programming in Pascal, we strongly suggest that the system be equipped with a printer. Pascal programs can become quite large, so it is desirable to be able to print out a listing of the program to work on it.

What Software Is Necessary?

Several software elements are required to program the IBM PC using Pascal. These are all provided with the IBM DOS, and the Pascal compiler package. Figure 1-2 is a diagram showing how these elements fit together. Now we'll discuss each one of these elements in detail.

Text Editor

We need a text editor to type in our Pascal source program. We used the WordStar program to create all of the programming examples in this book, but any other good word processor would do as well. You can also use IBM's text editor EDLIN to enter the source programs, although EDLIN is much more difficult to use than a real word processor. The main requirement is that the text editor produce a standard "text" file, PROG.PAS. You should also be careful that your word processor does not insert any non-standard characters in your file. WordStar, for example, has a "non-document" mode which does not insert "soft" spaces or carriage returns.

Pascal Compiler

The IBM Pascal system consists of several components which come with the compiler package when you buy it. There are two *phases* to the compiler itself. These compiler phases are actually two distinct programs. Each one must be executed separately. First, PAS1 is used to perform the

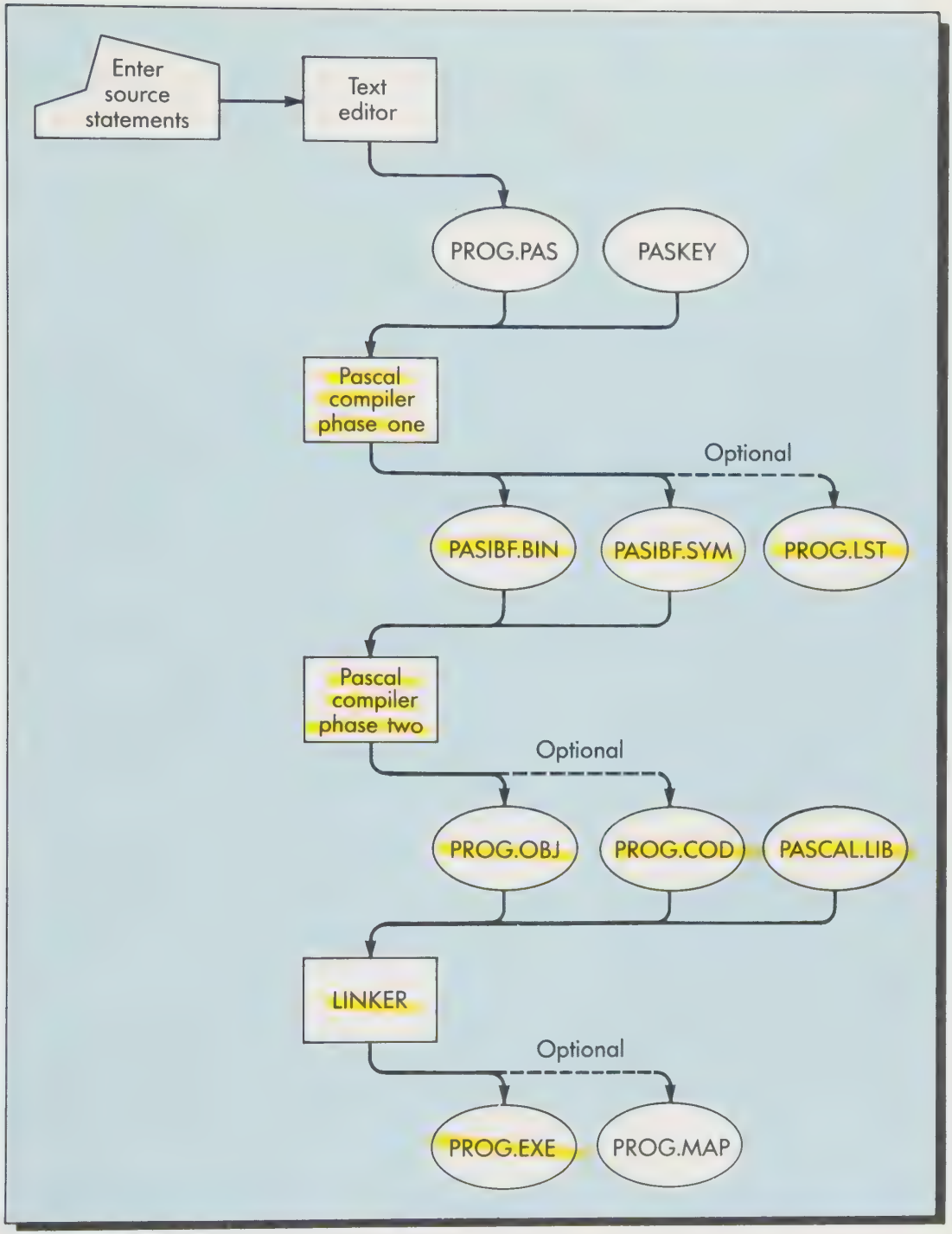


Figure 1-2. Elements of the Pascal system

initial compilation and detect errors. If no errors are detected, then PAS2 is executed to complete the compilation.

PAS1 — Scans the source program and generates PASIBF.BIN and PASIBF.SYM, the intermediate binary files used by PAS2. Also reports errors in the source program, and creates PROG.LST, the listing file.

PAS2 — Uses the intermediate binary files created by PAS1 to generate PROG.OBJ, the object file. Also creates PROG.COD, the optional listing of the object code.

The Pascal Library: PASCAL.LIB

The Pascal library is another file that comes with the compiler package. It contains all of the standard predeclared functions and procedures that can be used in a Pascal program. The library is used by the *linker* program (described next) during its generation of the executable program.

The Linker Program

Linking is the final step in the process of converting a source program written in Pascal into an executable form. The *Linker* is just another program that puts it all together. It uses the object file, PROG.OBJ, created by PAS2 of the compiler, along with the library file, PASCAL.LIB, to generate PROG.EXE, an executable program. There is also an optional file, PROG.MAP, that may be produced by the linker. This is a *map* of the linked program, showing the actual memory addresses of various parts of the final program.

The Linker program is part of the standard IBM DOS package, and is well documented in the DOS manual. Therefore, we will not go into detail on its operation.

Diskette Organization

If you are using a PC with two double-sided, double-density floppy disk drives, as we did writing this book, then you will want to consider the organization of files on the diskettes.

Purchasing the Pascal Compiler

When you purchase the Pascal compiler, you will receive three diskettes along with the manual. They are labeled:

PAS1	Pascal compiler (phase 1)
PAS2	Pascal compiler (phase 2)
PASCAL.LIB	Pascal library

These are single-sided diskettes, to support the users of PCs with single-sided drives. Of course, they are also readable by double-sided drives, so you can load them on your double-sided system.

The first thing to do, as IBM suggests, is to make a working copy of the IBM supplied diskettes. Then you can put the IBM disks away in your master archives so that you will always have the original. We used the DOS utility DISKCOPY to copy each diskette from drive A to drive B. We haven't touched the original IBM diskettes since then. They are safely stored, providing total backup of the Pascal system.

Realize that the copies you make of the diskettes will also be single-sided, even though you may have a double-sided disk drive. That's because the source diskettes were single-sided, and the DISKCOPY program produces an exact copy of the disk.

Once you have made copies of the three diskettes, you can go ahead with the actual system setup. You need to decide how to arrange the various elements of the Pascal system. Then, you can use the DOS command COPY to place those elements where you want them.

Typical System Layout

To adequately design a system layout, you must first decide exactly what you want the system to do. Once these goals are clear, you can determine your requirements, and the best way to satisfy them. Depending upon the type of application you are implementing, there will be some programs and files that you will want to have *on-line* all of the time.

For example, in our case, the primary application area for the IBM PC has been the creation of this book. To that end, we have used a word processing program, WordStar, to create the manuscript. Since the book is about Pascal, we have used the PC to create actual examples of Pascal programs. The source programs for these examples were also created using the WordStar program. Therefore, we needed to have WordStar on-line all the time. You will probably want your favorite editor or word processor available also.

```
A>dir
COMMAND  COM     4959   5-Ø7-82  12:ØØp
WS       COM    2Ø48Ø                  <---- WordStar
CHKDSK   COM     172Ø   5-Ø7-82  12:ØØp
ASTCLOCK COM      813   9-18-82         <---- Hardware clock rout.
XTALK    COM    14336  12-1Ø-82   3:14p <---- Communications prog.
DEBUG    COM     5999   5-Ø7-82  12:ØØp
WSMSGS   OVR    27264                  <---- WordStar
WSOVLY1  OVR    4Ø96Ø                  <---- WordStar
FILKQQ   INC     5347   8-Ø5-81        <---- File Control Block
FILUQQ   INC     1513   8-Ø5-81              interface files.
AUTOEXEC BAT      128   2-14-83   6:29p
XTALK    HLP    13952   7-15-82   1:44p <---- Communications prog.
IFIGXXXX TXT      64Ø   8-Ø3-83  11:22p
PASKEY           2816   8-Ø5-81        <---- Pascal predeclared
SAVEIMAG OBJ      342   8-Ø9-83  11:16a
DRAW     OBJ      433   7-27-83   3:37p
SAVEIMAG EXE     3456   8-Ø9-83  11:16a
GROUP             384   5-24-83  1Ø:38p
ANSWER            384   5-18-83   9:12a
BANDW    OBJ      123   7-27-83   1:Ø8p
COLORM   OBJ      124   4-28-83   9:53a
HIGHRES  OBJ      125   4-28-83   5:Ø4p
PORTIN   OBJ       66   4-22-83   4:58p
PORTOUT  OBJ       7Ø   4-26-83   8:Ø9p
SAVEIMAG ASM     384Ø   8-Ø9-83  11:15a
SETLF    EXE    2496Ø   5-Ø5-83   2:23p
COMPILE  BAT      128   4-29-83  12:Ø3p
IPASXXXX MAN      384   7-21-83   7:Ø3p
PLOT     OBJ       73   7-27-83   1:Ø7p
       3Ø File(s)


A>chkdsk

   322560 bytes total disk space
     9216 bytes in 2 hidden files
   195584 bytes in 31 user files
   117760 bytes available on disk

   131072 bytes total memory
   118672 bytes free
```

Table 1-1. Directory of system diskette

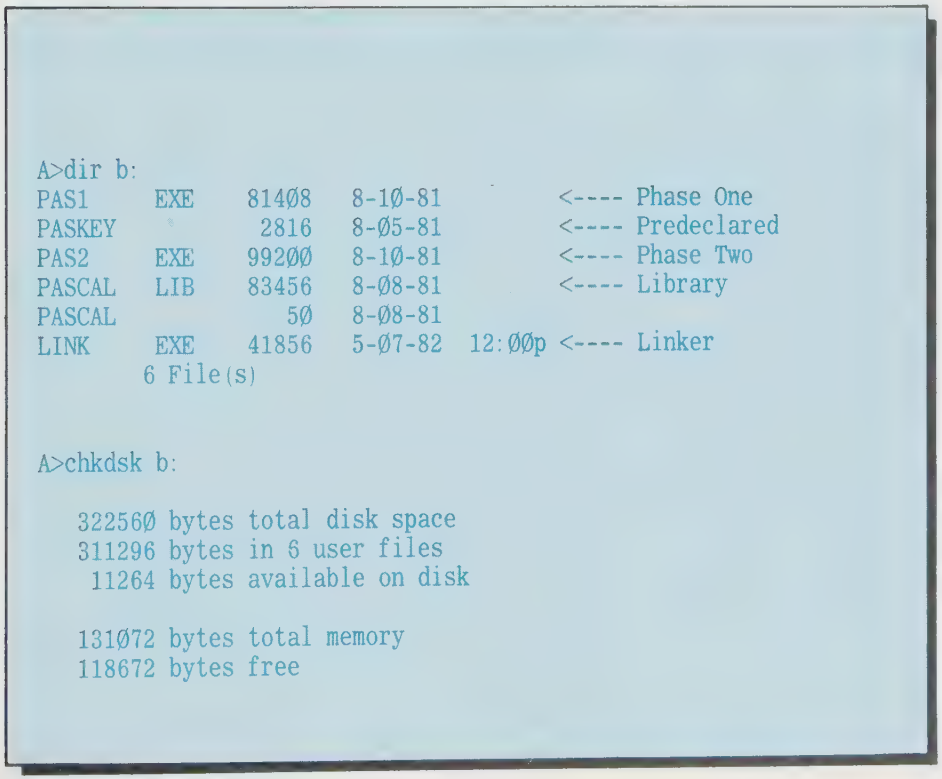
The next requirement is the Pascal compiler. You will need both phases of the compiler, the library file, and the linker program to be on-line all the time also.

Finally, you will need the typical system utilities such as CHKDSK to check the disk space available, DISKCOPY for making backups, and DEBUG for those "interesting" problems.

Oh yes! One more thing you'll need is some room on the disk to develop new programs. You'll be surprised by how much disk space is used in compiling one Pascal program.

As you can see from Tables 1-1 and 1-2, our approach to disk organization starts with considering the A drive as the system drive, and the B drive as the Pascal drive. We've put all our "system stuff" on the A drive. We've also located WordStar on the A drive. This leaves quite a bit of work space on the A drive, as indicated by the CHKDSK program. In fact, there are several files on our A drive that we will be using later in this book.

We have put all the Pascal elements on the B drive. Both phases of



```
A>dir b:
PAS1     EXE    814Ø8   8-1Ø-81          <---- Phase One
PASKEY           2816   8-Ø5-81          <---- Predeclared
PAS2     EXE    992ØØ   8-1Ø-81          <---- Phase Two
PASCAL   LIB    83456   8-Ø8-81          <---- Library
PASCAL            5Ø   8-Ø8-81
LINK     EXE    41856   5-Ø7-82  12:ØØp <---- Linker
        6 File(s)


A>chkdsk b:

   322560 bytes total disk space
   311296 bytes in 6 user files
    11264 bytes available on disk

   131Ø72 bytes total memory
   118672 bytes free
```

Table 1-2. Directory of Pascal diskette

the Pascal compiler, the library, auxiliary files, and the linker program are all loaded onto one diskette, leaving very little space left over. But that's all right, because all of our work in progress will be on the A drive.

Although the system organization described worked well for us, there are many other possible ways to organize your particular system to meet your needs. The point is that you must give some thought to the problem and come up with the best solution for your situation.

So now that we're organized, let's get down to the main event.

What Is a Pascal Program?

A Pascal *program* is a series of declarations and statements that are compiled into an object file by the Pascal compiler. Certain things are common to all Pascal programs. These are syntactic things, like the basic structure of the program, how to indicate a remark within the program, and so on.

Figure 1-3 presents the "general" form of a Pascal program. The intent is to illustrate briefly how the various parts of a Pascal program fit together. As we progress throughout the book, we will be exploring each of these parts in detail. For now, just relax and look at the big picture.

Figure 1-3. General Pascal form

```
{*****************************************************************
GENPAS.TXT      General Pascal Form.
-----------------------------------------------------------------
        This is the general form of a Pascal program, beginning
with the "Program declaration."
---- ----- ---------------------------------------------------
;
Program Prog_name (input,output,other_files);

 -- - - -- ---------------------------------------------------
                      Declarative Section.
---- -------  ---------------------------------------------------
;
Const
        const_name = value;
        *    *    *
        *    *    *
Type
        type_name = type_spec;
        *    *    *
        *    *    *
```

```
Var
        var_name,
        var_name,
        var_name,
                        :type_name;
        *    *    *
        *    *    *
Value
        var_name := init_value;
        *    *    *
        *    *    *

Function Func_name (param)     :type_name;
        Const
        Type
        Var
        Value
        Begin
                { Body of the function }
                    *    *    *
                    *    *    *
        end;

Procedure Proc_name (param     :type_name);
        Const
        Type
        Var
        Value
        Begin
                { Body of the procedure }
                   *    *    *
                   *    *    *

        end;
{
-------------------------------------------------------------------
                      Executable Section.
-------------------------------------------------------------------
}
Begin
        { Body of main program }
          *    *    *
          *    *    *
End.
{
-------------------------------------------------------------------
Figure 1-3.      GENPAS.TXT      General Pascal form
*******************************************************************}
```

Sections of a Pascal Program

There are two major sections to every Pascal program: the *declarative section* and the *executable section*. Sometimes the latter is referred to as the *program body*.

Everything is "declared" in Pascal. That's because the language is painstakingly thorough about making sure that every aspect of its process is spelled out. While seeming at first to be a lot of extra work, this requirement will force you to be a better programmer, and your programs will be more efficient and contain fewer errors.

Program Declaration

Every program must have a *Program declaration* statement as its first line. This tells the compiler that it is a PROGRAM being compiled, rather than a MODULE or IMPLEMENTATION. (In chapter 7, "Systems of Programs," we will discuss MODULEs and IMPLEMENTATIONs.) The PROGRAM declaration looks like this:

```
Program Progname (input,output);
```

This statement establishes the name of the program as "Progname" and lists the files that it will use. There are two special files in Pascal called *input* and *output*. These files correspond to the keyboard and display screen respectively. Any program that is going to input from the keyboard or output to the display screen must specify these two files in its program declaration. Of course, if your program is not going to use one or the other, then their names can be removed from the program declaration.

CONSTant Section

Actually, CONSTants may appear anywhere in the declarative section. Constants cannot change values when the program is run. We like to place them all at the very beginning of the declarative section, because they are often used in subsequent declarations of VARiables (discussed soon) to specify array sizes and subranges. CONSTant declarations look like this.

```
Const
    const_name = const_value;
```

TYPE Section

Special data *types* that will be used in the program are declared in the TYPE section. The types may be simply derivatives of the standard data

types, or completely custom "enumerated" types relevant to the application at hand.

Once the data type has been declared, it can be used in subsequent VARiable declarations. Here is the general TYPE declaration:

```
Type
     type_name = type_spec;
```

VARiable Section

The VARiable section is where any data element that is not a CONSTant is declared. These are the VARiable declarations. Variables may change values when the program is run. They may be arranged in any order, although we have a preference here as well. We like to declare all the variables of a certain data type together. For example, we would declare all the INTEGER types in a group, then all the CHAR types, BOOLEAN, REAL and so on. The general form is:

```
Var
     var_name  :type;

     var_name,
     var_name,
     var_name  :type;
```

VALUE Section

It is in the VALUE section that any initial *values* are assigned to their respective VARiables. This may include such things as starting a count at zero, or some other beginning value. This differs from the CONSTant section in that values assigned here are only initial values and may be changed during program execution. CONSTants, on the other hand, may not be changed by the program, and will retain the same value throughout program execution. The VALUE section looks like this.

```
Value
     var_name := init_value;
```

Functions and Procedures Section

Any *functions* and *procedures* that will be used by the program are defined in this section. They are not the standard functions and procedures which come automatically with the Pascal compiler, but functions and procedures that are defined here for the sole use of this program.

Functions and procedures are like little programs, included within a larger one. They can have their own declarative sections following the

function or procedure declaration statement. Each one also has an executable section that looks just like that of a program. Later, we'll devote an entire chapter (chapter 5) to studying functions and procedures.

Executable Section

The executable section is the body of the program. It contains all of the program statements. The executable section always starts with the BEGIN statement, and stops with the END statement and a period.

Program Remarks

Pascal can be a very "self-documenting" programming language. Not only are its data types well defined, and its statements logical and clear in their own right, but it is also possible to include a thorough commentary on the operation of the program. This is done with *comment* statements which are enclosed in braces ({}). These remarks are intermixed right with the program statements, and can even be on the same line. The compiler simply prints them out on the compilation listing. They do not occupy memory during execution of the program, the way REM statements do in an interpreted BASIC program. Comments can be many lines long: the only requirement is that they begin with a left brace ({) and end with a right brace (}).

How Is a Program Compiled?

Now that we've seen all the different elements of a Pascal program in general, let's get specific, and look at how we would go about actually compiling a small program. Remember that there are two phases for the compiler, plus the linker step at the end.

The Source Program: MAGIC.PAS

For our example, we'll use the Magic Number program. This is a simple little program that calculates a "magic number" by averaging the ASCII values for characters in the user's name. Figure 1-4 is a listing of the Pascal source statements for the program.

Figure 1-4. Magic number program

```
{*****************************************************************
MAGIC.PAS       Magic Number Program
-----------------------------------------------------------------
        This is an introductory Pascal program.  After the user
enters his or her name, the program computes a "magic number"
from the ASCII values of the characters in the name.
-----------------------------------------------------------------}

Program Magic (input,output);    { program declaration          }

{---------------------------------------------------------------
                    Declarative section.
---------------------------------------------------------------}

Const
        beep = Chr (7);          { beep the speaker             }

Var
        name                     { name as entered              }
                        :lstring (15);

        magic_num,               { magic number                 }
        length,                  { character count for name     }
        position                 { position marker in name      }
                        :integer;

Value
        magic_num := Ø;          { initialize magic number      }

{---------------------------------------------------------------
                Main program.  Input user's name.
---------------------------------------------------------------}
}
Begin

        Repeat
                Writeln;
                Write ('Hello!  Please sign in ----> ');
                Readln (name);
        Until name.len > Ø;

{---------------------------------------------------------------
              Compute and output the magic number.
---------------------------------------------------------------}

        length := Ord (name.len);
        For position := 1 to length Do
                magic_num := magic_num + Ord (name[position] );
```

```
        magic_num := magic_num Div length;
        Writeln (beep, 'Thanks ', name);
        Writeln ('Your magic number is ', magic_num);

End.
{
-----------------------------------------------------------------
Figure 1-4.       MAGIC.PAS       Magic number program
*****************************************************************}
```

Don't worry if you don't understand all the statements in this program. Our purpose is merely to show an example of the compilation process. Later we'll begin our discussion of the language itself.

First of all, Pascal source programs must have the DOS filename extension "PAS." Notice the program declaration right at the beginning after a multiple line comment. Then come the rest of the declarations, not many for this little program. There are no functions or procedures in this one, so we move right to the executable section. This starts with the BEGIN statement, and continues until the END statement.

To illustrate how well Pascal programs can be documented within the source file, we have included several forms of comments. We also use comments to provide headings for the different sections of the program.

Compiler Pass One

The first thing we do is to run phase one of the Pascal compiler. This is the program PAS1, which is resident on the B drive of our system. The screen below presents an example of running this phase. There are several options that must be specified, including whether or not to print the listing files MAGIC.LST and MAGIC.COD.

```
A>b:pas1
IBM Personal Computer Pascal Compiler
Version 1.ØØ (C)Copyright IBM Corp 1981
Source filename [.PAS]: magic          <-- file created with word processor
Object filename [MAGIC.OBJ]:           <-- default name
Source listing  [NUL.LST]: magic       <-- optional listing file
Object listing  [NUL.COD]: magic       <-- optional code file
 Pass One    No Errors Detected.
```

The Listing File: MAGIC.LST

During phase one of the compilation process, the compiler will display errors on the screen. Sometimes, if you're quick enough, you can see the problem right away. Usually though, it is desirable to print a listing of the entire compilation, which also includes the error messages. The listing is not printed during the compilation. Rather, the compiler creates an ASCII file on the disk with the same name as the program, but the extension "LST" instead of "PAS." This file can then be viewed using the TYPE command, or a word processor. The listing file is optional, and may be omitted from the compilation process, as we shall see.

Figure 1-5 is the compiler listing for the Magic Number program. One nice thing about the compiler listing is that the date and time are printed at the top of each page. This information can be compared to the date and time in the disk directory, to verify that the listing represents the current version of the program. Other than that, we have not found the listings particularly useful (except for correcting errors). The numbers on the left side of the listing represent line numbers, which are useful if you are using EDLIN, or another line-oriented text editor, to create your Pascal source files.

Figure 1-5. Compiler listing for MAGIC.PAS

```
*******************************************************************************
                                                                    Page   1
                                                                    Ø8-16-83
                                                                    11:Ø7:11
JG IC  Line#   Source Line       IBM Personal Computer Pascal Compiler V1.ØØ
    ØØ      1   {************************************************************
    ØØ      1
            2   MAGIC.PAS       Magic Number Program
            3   ------------------------------------------------------------
            3
            4           This is an introductory Pascal program.  After the user
            5   enters his or her name, the program computes a "magic number"
            5   from the ASCII values of the characters in the name.
            6
            7   ------------------------------------------------------------
            7
            8   }
    2Ø      9   Program Magic (input,output);    { program declaration }
           1Ø
           11   ------------------------------------------------------------
           11
```

```
                12                          Declarative section.
                13      ------------------------------------------------------------
                13
                14      }
    1Ø          15      Const
    1Ø          16              beep = Chr (7);             { beep the speaker }
                17
    1Ø          18      Var
    1Ø          19              name                        { name as entered }
    1Ø          2Ø                              :lstring (15);
                21
    1Ø          22              magic_num,                  { magic number }
    1Ø          23              length,                     { character count for name }
    1Ø          24              position                    { position marker in name }
    1Ø          25                              :integer;
                26
    1Ø          27      Value
    1Ø          28              magic_num := Ø;             { initialize magic number }
                29      {
                3Ø      ------------------------------------------------------------
                3Ø
                31                      Main program.  Input user's name.
                32      ------------------------------------------------------------
                32
                33      }
    1Ø          34      Begin
                35
    11          36              Repeat
    12          37                      Writeln;
    12          38                      Write ('Hello!  Please sign in ----> ');

MAGIC

                                                                        Page   2
                                                                        Ø8-16-83
                                                                        11:Ø7:13
JG IC  Line#    Source Line         IBM Personal Computer Pascal Compiler V1.ØØ
    12      39                      Readln (name);
    11      4Ø              Until name.len > Ø;
            41      {
            42      ------------------------------------------------------------
            43                  Compute and output the magic number.
            44      ------------------------------------------------------------
            45      }
    11      46              length := Ord (name.len);
    11      47              For position := 1 to length Do
    11      48                      magic_num := magic_num + Ord (name [position] );
            49
    11      5Ø              magic_num := magic_num Div length;
```

```
   11     51          Writeln (beep, 'Thanks ', name);
   11     52          Writeln ('Your magic number is ', magic_num);
          53
   ØØ     54   End.

Symtab    54   Offset Length   Variable
                    Ø     24   Return offset, Frame length
                    2     16   NAME                          :Array   Static
                   18      2   MAGIC_NUM                     :Integer Static
               Value
                   2Ø      2   LENGTH                        :Integer Static
                   22      2   POSITION                      :Integer Static

               Errors  Warns  In Pass One
                    Ø      Ø
```

Figure 1-5. MAGIC.LST Compiler listing for MAGIC.PAS

The Code File: MAGIC.COD

The code file is another optional file that is output by the compiler. It contains a sort of pseudo assembly language listing of the compiled program. We have not found this feature that useful, and normally omit this listing from our compilations. However, for completeness we have included the code file for our Magic Number program in Figure 1-6.

Figure 1-6. Code listing for MAGIC.PAS

```
Procedure/Function : MAGIC

    ** ØØØØØ1    PUSH      BP
    ** ØØØØØ2    MOV       BP,SP
    ** ØØØØØ4    SUB       BP,ØØØ6H
    ** ØØØØØ8    JB        $+6
    ** ØØØØØA    CMP       BP,STKHQQ
    ** ØØØØØE    JA        $+3
    ** ØØØØ1Ø    CALL      I4Ø92
    ** ØØØØ13    MOV       SP,BP
    ** ØØØØ15    ADD       BP,ØØØAH
    ** ØØØØ19    LCALL     INIFQQ
L36:
I5:
L37:
    ** ØØØØ1E    MOV       DX,@@OUTFQQ
    ** ØØØØ21    PUSH      DX
    ** ØØØØ22    LCALL     WTLFQQ
```

```
L38:
    ** 000027    MOV         DX,@@OUTFQQ
    ** 00002A    PUSH        DX
    ** 00002B    MOV         DX,001DH
    ** 00002E    PUSH        DX
    ** 00002F    MOV         DX,@@<const>+48
    ** 000032    PUSH        DX
    ** 000033    MOV         DX,7FFFH
    ** 000036    PUSH        DX
    ** 000037    PUSH        DX
    ** 000038    LCALL       WTSFQQ
L39:
    ** 00003D    MOV         DX,@@INPFQQ
    ** 000040    PUSH        DX
    ** 000041    MOV         DX,000FH
    ** 000044    PUSH        DX
    ** 000045    MOV         DX,@@NAME
    ** 000048    PUSH        DX
    ** 000049    LCALL       RTTFQQ
    ** 00004E    MOV         DX,@@INPFQQ
    ** 000051    PUSH        DX
    ** 000052    LCALL       RTLFQQ
L40:
I3:
    ** 000057    MOV         DX,NAME
    ** 00005B    XOR         DH,DH
    ** 00005D    CMP         DX,01H
    ** 000060    JB          I5
I4:
L46:
    ** 000062    MOV         AX,NAME
    ** 000065    XOR         AH,AH
    ** 000067    MOV         LENGTH,AX
L47:
    ** 00006A    MOV         DX,LENGTH
    ** 00006E    MOV         [BP].FAH,DX
    ** 000071    CMP         [BP].FAH,01H
    ** 000075    JL          I7
    ** 000077    MOV         POSITI,0001H
I8:
L48:
    ** 00007D    PUSH        POSITI
    ** 000081    XOR         DX,DX
    ** 000083    PUSH        DX
    ** 000084    MOV         DX,000FH
    ** 000087    PUSH        DX
    ** 000088    LCALL       RCIEQQ
    ** 00008D    XCHG        AX,DI
    ** 00008E    MOV         DX,NAME[DI]
```

```
     ** ØØØØ92    XOR       DH,DH
     ** ØØØØ94    ADD       DX,MAGIC_
     ** ØØØØ98    JNO       $+3
     ** ØØØØ9A    CALL      I4Ø94
     ** ØØØØ9D    XCHG      AX,DX
     ** ØØØØ9E    MOV       MAGIC_,AX
I6:
     ** ØØØØA1    MOV       AX,POSITI
     ** ØØØØA4    INC       AX
     ** ØØØØA5    MOV       POSITI,AX
     ** ØØØØA8    DEC       AX
     ** ØØØØA9    CMP       AX,[BP].FAH
     ** ØØØØAC    JNE       I8
I7:
L5Ø:
     ** ØØØØAE    MOV       AX,MAGIC_
     ** ØØØØB1    CWD
     ** ØØØØB2    TEST      LENGTH,FFFFH
     ** ØØØØB8    JNE       $+3
     ** ØØØØBA    CALL      I4Ø91
     ** ØØØØBD    IDIV      LENGTH
     ** ØØØØC1    MOV       MAGIC_,AX
L51:
     ** ØØØØC4    MOV       DX,@@OUTFQQ
     ** ØØØØC7    PUSH      DX
     ** ØØØØC8    MOV       DX,ØØØ7H
     ** ØØØØCB    PUSH      DX
     ** ØØØØCC    MOV       DX,7FFFH
     ** ØØØØCF    PUSH      DX
     ** ØØØØDØ    PUSH      DX
     ** ØØØØD1    LCALL     WTCFQQ
     ** ØØØØD6    MOV       DX,@@OUTFQQ
     ** ØØØØD9    PUSH      DX
     ** ØØØØDA    MOV       DX.ØØØ7H
     ** ØØØØDD    PUSH      DX
     ** ØØØØDE    MOV       DX,@@<const>+78
     ** ØØØØE1    PUSH      DX
     ** ØØØØE2    MOV       DX,7FFFH
     ** ØØØØE5    PUSH      DX
     ** ØØØØE6    PUSH      DX
     ** ØØØØE7    LCALL     WTSFQQ
     ** ØØØØEC    MOV       DX,@@OUTFQQ
     ** ØØØØEF    PUSH      DX
     ** ØØØØFØ    MOV       DX,ØØØFH
     ** ØØØØF3    PUSH      DX
     ** ØØØØF4    MOV       DX.@@NAME
     ** ØØØØF7    PUSH      DX
     ** ØØØØF8    MOV       DX,7FFFH
     ** ØØØØFB    PUSH      DX
```

```
    ** ØØØØFC    PUSH      DX
    ** ØØØØFD    LCALL     WTTFQQ
    ** ØØØ1Ø2    MOV       DX,@@OUTFQQ
    ** ØØØ1Ø5    PUSH      DX
    ** ØØØ1Ø6    LCALL     WTLFQQ
L52:
    ** ØØØ1ØB    MOV       DX,@@OUTFQQ
    ** ØØØ1ØE    PUSH      DX
    ** ØØØ1ØF    MOV       DX,ØØ15H
    ** ØØØ112    PUSH      DX
    ** ØØØ113    MOV       DX,@@<const>+86
    ** ØØØ116    PUSH      DX
    ** ØØØ117    MOV       DX,7FFFH
    ** ØØØ11A    PUSH      DX
    ** ØØØ11B    PUSH      DX
    ** ØØØ11C    LCALL     WTSFQQ
    ** ØØØ121    MOV       DX,@@OUTFQQ
    ** ØØØ124    PUSH      DX
    ** ØØØ125    PUSH      MAGIC_
    ** ØØØ129    MOV       DX,7FFFH
    ** ØØØ12C    PUSH      DX
    ** ØØØ12D    PUSH      DX
    ** ØØØ12E    LCALL     WTIFQQ
    ** ØØØ133    MOV       DX.@@OUTFQQ
    ** ØØØ136    PUSH      DX
    ** ØØØ137    LCALL     WTLFQQ
L54:
    ** ØØØ13C    LEA       SP,[BP].FCH
    ** ØØØ13F    POP       BP
    ** ØØØ14Ø    LRET
I4Ø94:
    ** ØØØ141    LCALL     SAOGQQ
I4Ø93:
    ** ØØØ146    LCALL     UAOGQQ
I4Ø92:
    ** ØØØ14B    LCALL     SOVGQQ
I4Ø91:
    ** ØØØ15Ø    LCALL     SDZGQQ
I4Ø9Ø:
    ** ØØØ155    LCALL     UDZGQQ
```

Figure 1-6. MAGIC.COD Code listing for MAGIC.PAS

****

Compiler Pass Two

If there were no errors encountered during the first phase of the Pascal compilation, then we may proceed with the second phase. The

second phase of the compiler uses the intermediate binary files created by the first phase to create an object file with the same name as the source file, except the extension is "OBJ." It is this object file that is turned into an executable program file by the linker. The screen below shows how the second compiler phase looks when executed.

```
A>b:pas2

Code Area Size = #Ø15A  (346)
Cons Area Size = #ØØ6B  (1Ø7)
Data Area Size = #ØØ18  (24)

Pass Two    No Errors Detected.
```

As you can see, there really isn't that much to running the second phase of the compiler. If the first phase was successful, and there were no compilation errors, then everything is automatically set up to run the second phase.

The compiler displays the memory requirements for the program. These are somewhat deceiving, since there is a large runtime module that surrounds the actual Pascal program. The final product (the EXE file produced by the linker) will require over 20K bytes of disk, in addition to what PAS2 tells you the size of the program is.

The Linker

The linker is the last step in the process of making an executable program from the Pascal source statements. There are several options with the linker program that we must specify by answering the prompts. First of all, we need to specify the name of the OBJ file that has just been created by the second phase of the compiler. Next, we can accept the *default* file name for the executable program by simply pressing Enter. A *memory map* listing is an option here, and we have elected to print it. Normally, we omit this listing by pressing Enter, but we'll print one this time so you can see what it looks like.

Finally, the linker asks if there are libraries to be involved in this linkage. The linker already knows that it is linking a Pascal program, and that it must use the library of Pascal routines, PASCAL.LIB. However, since this library is located on the B drive (at least the way our system is set up), and the A drive is currently specified as the default, we must tell the linker to look for the library on the B drive. The next screen is an example of running the linker.

```
A>b:link

IBM Personal Computer Linker
Version 1.1Ø (C)Copyright IBM Corp 1982

Object Modules [.OBJ]: magic      <-- file created by PAS2
Run File [MAGIC.EXE]:             <-- default name
List File [NUL.MAP]: magic        <-- optional load map file
Libraries [.LIB]: b:              <-- library is on B drive
```

```
 Start  Stop   Length  Name                    Class
 ØØØØØH ØØ159H Ø15AH   MAGIC                   CODE
 ØØ16ØH ØØ485H Ø326H   MISGQQ                  CODE
 ØØ486H Ø14D2H 1Ø4DH   FILFQQ_CODE             CODE
 Ø14D4H Ø1AAEH Ø5DBH   ORDFQQ_CODE             CODE
 Ø1ABØH Ø1ABØH ØØØØH   INIXQQ                  CODE
 Ø1ABØH Ø1B76H ØØC7H   ENTXQQ                  CODE
 Ø1B78H Ø1FØ8H Ø391H   STRFQQ_CODE             CODE
 Ø1FØAH Ø2847H Ø93EH   ERREQQ_CODE             CODE
 Ø2848H Ø38EAH 1ØA3H   FILUQQ_CODE             CODE
 Ø38ECH Ø39DBH ØØFØH   MISYQQ_CODE             CODE
 Ø39DCH Ø4249H Ø86EH   CODCQQ_CODE             CODE
 Ø424AH Ø43BØH Ø167H   PASUQQ_CODE             CODE
 Ø43B2H Ø43DFH ØØ2EH   MISOQQ_CODE             CODE
 Ø43EØH Ø459ØH Ø1B1H   HEAHQQ_CODE             CODE
 Ø4592H Ø46B2H Ø121H   UTLXQQ_CODE             CODE
 Ø46B4H Ø4759H ØØA6H   MISHQQ_CODE             CODE
 Ø476ØH Ø476ØH ØØØØH   HEAP                    MEMORY
 Ø476ØH Ø476ØH ØØØØH   MEMORY                  MEMORY
 Ø476ØH Ø495FH Ø2ØØH   STACK                   STACK
 Ø496ØH Ø5Ø47H Ø6E8H   DATA                    DATA
 Ø5Ø5ØH Ø578DH Ø73EH   CONST                   CONST
 Ø579ØH Ø579ØH ØØØØH   ??SEG

Program entry point at Ø1AB:ØØØØ
```

Figure 1-7. Memory map for MAGIC.PAS

The Map File: MAGIC.MAP

This is the optional listing that can be obtained from the linker. It is a map of the program, as it will reside in memory during execution. The start and stop memory addresses for each of the routines involved with the program are listed. Also listed is the *program entry point*. This is where execution will begin when the program is run. This listing, again, is not particularly useful, although if you're interested in the inside structure of Pascal, you may find it indispensable. The map file is shown in Figure 1-7.

Files Related to the Compilation

There are a number of files that are associated with every compilation. We have included all of them with the example of the Magic Number program, so that you would have an understanding of their nature. Table 1-3 is a directory listing that shows all of the files associated with compilation and linking of the Magic Number program.

Using a Batch File

As you saw in the previous example, there is quite a bit involved in taking a Pascal source program all the way through the compilation and linking process. This is mainly due to the number of options offered by

```
A>dir magic.*
MAGIC    PAS     192Ø   8-15-83  12:35p
MAGIC    LST     4232   8-16-83  11:Ø7a
MAGIC    MAP     128Ø   8-16-83  11:1Øa
MAGIC    EXE    24576   8-16-83  11:1Øa
MAGIC    OBJ     111Ø   8-16-83  11:Ø8a
MAGIC    COD     4763   8-16-83  11:Ø8a
       6 File(s)
```

Table 1-3. Files related to compilation

the compiler, and the necessity to enter file names for the object file, listing file, code file, executable file, and so forth. If you are going to be doing any amount of Pascal development, then you will want to set up a DOS *batch file* to handle some of these options automatically. (If you're not familiar with batch files, you can read all about them in the *IBM Personal Computer Disk Operating System* manual.)

We have created a batch file called "COMPILE.BAT," which contains all of the DOS commands necessary to invoke the compiler phases and the linker. This file also calls the DOS routine TIME, to get the start and stop times for the compilation from the system clock. It specifies automatically which of the optional files should be generated during the compilation and linking processes. Figure 1-8 is a listing of the batch file.

The symbol "%1" is a *pseudo-variable* for the batch file. Whatever is typed in on the same line with the DOS command to run the batch file (namely the source file name without the PAS), will be assigned to this pseudo-variable. Then, that name is passed on to the rest of the batch file. In this way, all we have to type to compile and link a program is:

```
A>compile progname
```

```
time                                    <---- Start time
b:pas1 %1.pas %1.obj %1.lst nul.cod     <---- Phase One
b:pas2                                  <---- Phase Two
b:link %1,%1.exe,nul,b:pascal.lib       <---- Linker
time                                    <---- Stop time
```

Figure 1-8. Batch file for compile and link

To illustrate this, we will compile the Magic Number program using the batch file approach. Here is how the screen looks during the compilation.

```
A>compile magic ENTER ENTER   <— double ENTER for TIME command

A>time
Current time is 12:Ø3:37.66
Enter new time:

A>b:pas1 magic.pas magic.obj magic.lst nul.cod   <— command line

IBM Personal Computer Pascal Compiler
Version 1.ØØ (C)Copyright IBM Corp 1981
 Pass One    No Errors Detected.

A>b:pas2   <— command line

 Code Area Size = #Ø15A   (346)
 Cons Area Size = #ØØ6B   (1Ø7)
 Data Area Size = #ØØ18   (24)

 Pass Two    No Errors Detected.

A>b:link magic,magic.exe,nul,b:pascal.lib   <— command line

IBM Personal Computer Linker
Version 1.1Ø (C)Copyright IBM Corp 1982

A>time
Current time is 12:Ø5:33.17
Enter new time:
```

Notice that we have indicated pressing Enter twice after typing the command to execute the batch file. This is because the first thing that happens is a call to the DOS routine TIME to get the starting time. Unfortunately, the TIME routine also expects the user to either enter a new time, or press Enter. Since all the keyboard I/O is buffered through the DOS, it is not necessary to wait for the TIME routine to display the prompt "Enter new time."

The command line for the compiler phase one, as shown in the screen display, specifies that a compiler listing file be created called MAGIC.LST, but that no code file be created. This is indicated by the "nul.cod" in the command line.

No parameters are necessary for PAS2.

The command line for the linker program specifies the name of the executable file. It also tells the linker not to print a map file. This is done by the "nul" value following the EXE file name. Finally, we have specified the Pascal library (located on the B drive) to be used during the linking process.

Summary

We have now seen a smattering of Pascal. We know the difference between a compiled language and an interpreted one. We have our diskettes organized for optimum efficiency, and we know all about source files (.PAS), object files (.OBJ), and executable files (.EXE).

Pascal is a special kind of programming language: one suitable for quite complex applications. It is not as easy a language to work with as BASIC, but it has many advantages over BASIC in certain situations. You'll get used to the compilation process, and the more you use it, the faster you will become at getting programs working.

Exercises

1. What are the advantages of using a compiled language such as IBM Pascal?

2. What are the two principal sections of a Pascal program?

3. Why do programs need to be linked?

Solutions

1. Compiled programs run much faster than interpreted programs. They can also include more detailed comments since the comments won't be part of the final program.

2. The declarative section and the executable section (or program body) are the main sections of a Pascal program.

3. The object program produced by the compiler (PAS2) must be combined with portions of the library, PASCAL.LIB, to be executed. This is done by the linker program.

2

Simple Data Types

Concepts

Data elements
Declarations
Data types
Type compatibility
Arithmetic expressions
Operators
Boolean operators
Evaluation of an expression
Simple functions

Pascal Keywords

TYPE, CONST, VAR, VALUE, INTEGER, MAXINT, DIV, MOD, WORD, MAXWORD, BYTE, CHAR, CHR, BOOLEAN, TRUE, FALSE, REAL, STRING, LSTRING, .LEN, TRUNC, ROUND, FLOAT, ABS, SQR, SORT, SIN, COS, ARCTAN, EXP, LN, CONCAT

One common element of all computer programs, no matter what language they are written in, is that they process information. Whether it be the simplest of game programs, or a real "number crunching" application, there will be some manipulation of *data elements*. In order to process data elements, the program must be able to set aside a space in memory for each piece of data. The number of bytes that are set aside, and how they are used, are determined by the TYPE of data element. The purpose of this chapter is to introduce "simple" data types and show how they are used in a program.

As we discuss data types you will also become familiar with the structure of simple Pascal programs.

Declaring Data Elements

The term *variable* is used in the programming community to describe the elements of data that will be used in a program. Actually, sometimes these data elements are not variables, but *constants* that will never change throughout the execution of the program. So, "variable" is really a misnomer, and Pascal (unlike many lesser languages) will have nothing to do with such imprecision. If a variable is really a constant, then it is declared so in Pascal.

Pascal Is Fussy

Pascal is one of the fussiest programming languages when it comes to dealing with variables. Many other programming languages (BASIC especially) are a lot less demanding of precise use of variables. In some forms of BASIC, there are only two different types of variables: numeric and string. BASIC usually differentiates between them by requiring the programmer to append a dollar sign "$" to the end of the name assigned to the string variables, as in the BASIC statement:

```
1Ø LET A$ = "PROGRAM NAME"
```

And that's it! No further specification of data types need be made. BASIC is very understanding and rather casual in its approach to program variables and constants.

As we saw in the last chapter, in Pascal all data elements must be declared in the beginning part of a Pascal program, before any executable statements appear. True variables are declared within the VAR section. Each variable identifier name and its type are declared. If there is to be a constant involved as a data element, then it is declared and defined in the CONST section. Pascal will not allow a constant to be altered by the program during execution.

Assignment

Assignment is the process by which a value is "assigned" to a variable in a VALUE statement or during the course of program execution. This is written differently in Pascal than in most other programming languages. In the BASIC example above, we simply used the equal sign (=) to indicate value assignment. In Pascal we must distinguish between "assignment" and "equality." The equal sign means *equality* in Pascal.

This is known as a condition, and is often involved in program control, which we'll be studying in the next chapter. To indicate assignment we must use a combination of the colon and the equal sign (:=). For instance, here's an assignment statement that assigns the value 10 to the variable identifier *discount*.

```
discount := 10;
```

Names in Pascal

IBM Pascal is very flexible in the names or *identifiers* you may use for variables. They must start with a letter (A-Z), but may contain up to 30 other letters or digits (0-9). To improve readability, you may use the underscore character (_) in the middle of a variable name to separate words within the name. The compiler makes no distinction between upper and lower case characters so that "FIRST_TRY," "First_Try," and "first_try" all refer to the same variable. These generous rules for names apply not only to variables, but to constants, and programs, as well.

Pascal Declarations

As we mentioned, Pascal differentiates between several types of variables that might be used in a program. All of them must be declared at the beginning of the program before any processing is done. The declarations are usually grouped together to make the program easier to read, with these Pascal keywords heading each division.

CONST	constant data elements are declared along with their values.
TYPE	declares a data type.
VAR	declares a variable.
VALUE	sets initial value of a variable.

We will see some of these keywords appear in almost every program in this book. Pascal makes sure that when a data element is used in a program, its use is compatible with the variable's type. This is done at two levels in IBM Pascal: first by the compiler and then by the runtime debug options.

There are so many different data types in Pascal that we have divided our discussion of them into two chapters. In this chapter we will discuss the variable types INTEGER, REAL, WORD, BYTE, CHAR, BOOLEAN, STRING, and LSTRING. We call these *simple* variable types, as compared to *structured* variable types which are discussed in chapter 4.

Pascal also allows the programmer to define a custom variable type for a particular application. These are called *enumerated types*, and can be used for handling data that consist of a finite set, such as a deck of cards, the primary colors, days of the week, and so on.

When a programming situation requires that the value a variable takes on be bounded between some limits during execution, Pascal does a great job. A variable can be declared to be valid only within a *subrange* of its natural type. Then, when the program is executed, any operation that results in a value outside the subrange produces an error.

So, let's take a look at each of the different variable types. We'll see how each is declared to the compiler, how it might be used within the body of a program, and how the variable is actually represented in memory.

As we discuss TYPEs in this chapter, we'll also be introducing a number of different Pascal programs as examples. Thus as you learn about data types you will also be absorbing the fundamentals of Pascal programming. Don't worry if every program statement is not clear to you. Type in the example programs, compile them (as explained in chapter 1), and try them out. This will give you a sense of what Pascal is all about, and eventually all the details will be explained.

Integer Type

The integer is probably the most common of data types. It is encountered in just about every computer programming language in existence. That's probably because of the simple way an INTEGER is usually represented in memory. In the IBM PC, an INTEGER occupies two consecutive bytes of memory, forming a 16-bit binary value, as shown in Figure 2-1. The first bit in the most significant byte of the INTEGER

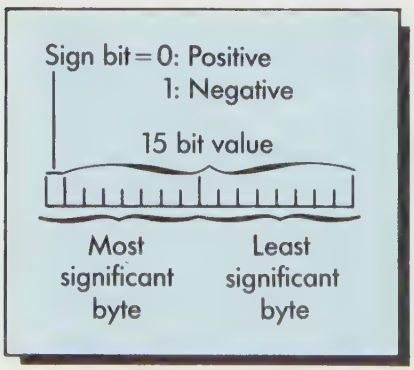


Figure 2-1. Internal representation of INTEGER type

type is used to indicate the arithmetic *sign* of the INTEGER.

'0' – indicates a positive number

'1' – indicates a negative number

This means that there are only 15 bits remaining in the INTEGER to represent its value. The largest value is 2**15 – 1, or 32,767. So an INTEGER can take on any value from –32767 to +32767. Since it may vary between computers, this limit has a name in Pascal. It is called MAXINT, and is a predeclared constant. (Note that –32768 is not a valid INTEGER value in Pascal, as it often is in other languages.)

Age Calculation Program

Here are some examples of the INTEGER declaration and the use of INTEGER types.

Figure 2-2. Age calculation program in years

```
{*************************************************************}
AGECALC.PAS     Calculate Age in Years.
{-------------------------------------------------------------}
{
Program Age_calc (input,output);
{
{-------------------------------------------------------------}
                    Declaration Division
{-------------------------------------------------------------}
{
Var
        age_in_years,
        year_of_birth,
        current_year
                              :integer;
{
{-------------------------------------------------------------}
          Main program.  Prompt for and input years.
{-------------------------------------------------------------}
{
Begin
        Write ('Enter current year ----> ');
        Readln (current_year);
        Write ('Enter year of birth ---> ');
        Readln (year_of_birth);
```

```
------------------------------------------------------------
            Calculate and output the age in years.
------------------------------------------------------------
;
        age_in_years := current_year - year_of_birth;

        Writeln ('Age in years is ', age_in_years);
End.
;
------------------------------------------------------------
```

Figure 2-2. AGECALC.PAS Age calculation program in years

The "Age_calc" program, shown in Figure 2-2, is a "safe" application of the INTEGER type, in that none of the data elements which are required in the program will exceed the MAXINT value. For the sake of simplicity, all three of the variables are declared as INTEGERS with one VAR statement. In the body of the program, some simple arithmetic calculates the age in years. Of course we remember to use the combination symbol (:=) to indicate the assignment of the value of the expression to the variable "age_in_years."

Before we go any further in this program, we should explain the statements that Pascal uses for simple input and output.

WRITELN and WRITE

WRITELN (pronounced "Write Line") is like a PRINT statement in BASIC. WRITE is like a PRINT statement in BASIC followed by a semicolon to prevent the automatic carriage return linefeed from taking place.

Examine the statement

```
Write ('Enter current year ----> ');
```

As you can guess, this causes

```
Enter current year ---->
```

to be displayed on the screen. Anything you want to display on the screen is placed in the parentheses following the WRITE or WRITELN.

If what you want to display is a *string*, or group of characters, which is to appear exactly as in your program, then you enclose it in single quotes ('), as shown in the example above. This is similar to enclosing strings in double quotes (") in BASIC PRINT statements. (We'll discuss strings in more detail later.)

On the other hand, if what you want to display is the value of some variable, then don't enclose it in single quotes. In the statement

```
Writeln ('Age in years is ', age_in_years);
```

the program will first display the string "Age in years is" and then display whatever number it has calculated and assigned to the integer variable "age_in_years."

READLN

READLN (pronounced "Read Line") reads a number (or in the case of string variables, a line of characters) that you have typed on the keyboard. The number (or string) must be terminated by pressing Enter. It is then assigned to the variable in parentheses following the READLN.

READLN is a good bit like BASIC's INPUT statement. In the program above, the statement

```
Readln (current_year);
```

will cause the program to wait for an integer to be typed at the keyboard. Once you have typed in the number and pressed ENTER, the value will be assigned to the variable "current_year."

We'll see many more examples of WRITE, WRITELN and READLN as we move along.

Month Age Calculation Program

Figure 2-3 provides an example of calculating age in months. In this example, the program needs to have more information in order to be more specific in the age calculation. Two additional INTEGER type variables are declared to represent both the "current_month" and the "month_of_birth." Notice that the expression for calculating the age is a little more complex. The three arithmetic operations "+", "−", and "*" are all used within the same statement. Also notice the appearance of the constant "12" in the expression. This is of course the number of months in a year, a factor we need to calculate age in months. The number 12 is really just another data element required by the program. When it appears like this in the middle of an expression, it's called an *implied constant*. It's similar to presenting a string as a *literal constant* enclosed in single quotes. The compiler will know what we want here. (Bet you thought you would have to declare all of your constants too.)

Figure 2-3. Age calculation program in months

```
{*****************************************************************
MONTHAGE.PAS    Calculate Age in Months.
-----------------------------------------------------------------
        This program calculates age in months, given the
current month and year, as well as the month and year of birth.
-----------------------------------------------------------------
}
Program Monthage (input,output);
{
-----------------------------------------------------------------
                    Declarative division.
-----------------------------------------------------------------
}
Var
        age_in_months,
        month_of_birth,
        year_of_birth,
        current_month,
        current_year
                            :integer;
{
-----------------------------------------------------------------
                        Main program.
-----------------------------------------------------------------
}
Begin
        Write ('Enter current month and year -------> ');
        Readln (current_month, current_year);
        Write ('Enter month and year of birth ------> ');
        Readln (month_of_birth, year_of_birth);
{
-----------------------------------------------------------------
             Calculate and output age in months.
-----------------------------------------------------------------
}
        age_in_months :=
                12 * (current_year - year_of_birth)     +
                     (current_month - month_of_birth);

        Writeln ('You are ', age_in_months, ' months old.');

End.
{
-----------------------------------------------------------------
Figure 2-3.      MONTHAGE.PAS      Age calculation program in months
*****************************************************************}
```

Evaluating an Expression

Let's take a look at the way that the expression will be evaluated. Notice the use of parentheses to group the elements of the expression to control the order of its evaluation. Without parentheses, there is a default "pecking order" among the arithmetic operators. Functions are evaluated first. Then come multiplication and division, and finally addition and subtraction. Let's use some arbitrary values.

Assume:

```
current_month       := 5;
current_year        := 1983;
month_of_birth      := 4;
year_of_birth       := 1945;
```

So, plugging these values into the expression:

```
age_in_months  := 12 * (1983 - 1945) + (5 - 4);
               := 12 * 38 + (5 - 4);
               := 456 + 1;
               := 457;
```

The first set of parentheses cause the subtraction of the years to occur before the result is multiplied by the constant 12. Once the grouped subtraction has been performed, the multiplication is given priority before the remainder of the expression is evaluated. The second set of parentheses is more for clarity in the program listing than for evaluation control.

Let's take one more example. Assume:

```
month_of_birth      := 10;
year_of_birth       := 1982;
```

Now,

```
age_in_months  := 12 * (1983 - 1982) + (5 - 10);
               := 12 * 1 + (5 - 10);
               := 12 + (-5);
               := 7;
```

Exercise in Reality

And, just to make this a realistic book, let's throw in an error. Suppose we transposed the digits in the current year when we assigned it and instead of 1983, entered 9183. Now when we plug in the numbers:

```
age_in_months  := 12 * (9183 - 1982) + (5 - 1Ø);
               := 12 * 72Ø1 + (5 - 1Ø);
               := 86412 + (-5);
               := 864Ø7;
```

This would result in a runtime error since the value calculated in the expression exceeds the MAXINT value of 32,767. This is a trivial example of a feature that makes Pascal very desirable for those programming applications where it is essential that no errors trickle through the process without being caught.

Integers Are Versatile

So, the INTEGER variable is useful for many common data requirements. It has a range large enough to represent simple data elements. Later, we will see how integers are used in program control logic. Generally, operations involving INTEGERs will execute faster than those involving the REAL types described later in this chapter. That is because there are actual machine level instructions which add, subtract, and multiply 16-bit quantities. This means that arithmetic operations on INTEGER types will only require one machine instruction, while operations on REAL types will require many instructions.

The DIV and MOD Functions

You may have noticed that we have not shown any examples of INTEGER division yet. That's because division is not accomplished with the usual "/" operator. The "/" operator is reserved for REAL types. To divide INTEGERs we must use the DIV function. Since this function produces an INTEGER result, in many cases it will return a truncated version of the actual quotient. In other words, any "remainder" (or digits to the right of the decimal place) will be dropped. For example:

20 div 5 = 4

but,

20 div 3 = 6

The MOD function is similar to DIV, except that it returns the remainder of the divide operation, rather than the quotient. Some examples of the MOD function are:

16 mod 2 = 0

35 mod 10 = 5

1 mod 10 = 1

So, that's not too hard. In fact, these functions can prove very useful, as you will see in some of our example programs.

Dollars and Cents Quantities

In some cases, it may be desirable to represent a dollars and cents amount as an INTEGER to take advantage of the processing speed of the INTEGER arithmetic. This can be done within limits if you simply imagine the decimal point between the second and third digits, counting from the right, and then require that the value of the dollar amount shall never be greater than 327.67.

Of course, when the dollars and cents amount exceeds 327.67, we are forced to use the REAL type data element (discussed later in this chapter). Even with the REAL type, there are limitations. As we shall see, there is also a "precision" limitation on REAL types of about 6 digits of accuracy. This will lead to round-off errors, especially when performing division operations.

WORD Type

Sometimes a programming application will require the use of numbers larger than MAXINT (32,767) but will not need negative values for these numbers. In such a case it would be nice to utilize all 16 bits of an INTEGER variable to represent an unsigned value. This situation is handled by the WORD type shown in Figure 2-4. It can have a range from 0 to 65,535. Pascal also has a name for this limit. It is a constant called MAXWORD, and can be used just like MAXINT. Notice that negative values are not allowed since there is no sign bit.

The WORD type variable is also useful for dealing with 16-bit quantities of unknown meaning. These could be integers, characters, or

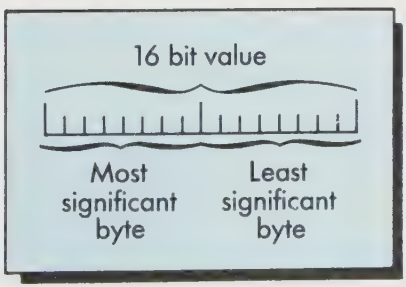


Figure 2-4. Internal representation of WORD type

maybe part of a record or array. Treated as WORDs, their 16-bit configuration will remain undisturbed, and range errors involved with INTEGER manipulation can be avoided.

Memory Address Manipulation

An example of the use of WORD types is manipulating memory addresses. A powerful feature of IBM PC Pascal is its ability to deal with any location, or address, in the segmented memory environment. The addresses all have two parts, the *segment* and *relative* parts. (In Pascal nomenclature, "segment" and "relative" addresses are synonymous with "segment" and "offset" in the rest of the IBM literature.) These are both represented within the computer as 16-bit unsigned values, making the WORD type a natural for the application. This provides the Pascal programmer with the ability to access actual locations anywhere in the memory of the IBM PC. Thus Pascal programs can interface to the operating system and other areas of the machine. This ability is similar to the "PEEK" and "POKE" statements of some BASICS, but much more versatile.

We will demonstrate this ability in several examples later in this book.

WORD and INTEGER Compatibility

In general, the WORD type variable can be used like an integer. Arithmetic operations can be performed on WORD variables with the understanding that a WORD variable can not take on a negative value since it is treated as a 16-bit unsigned quantity. Therefore, INTEGER and WORD variables should not be mixed in an expression. The Pascal compiler will issue a warning if they are and the resulting evaluation of the expression will depend on the order of the different variables in the expression.

INTEGER and WORD types are not "assignment compatible" either. That means that you can not directly assign one to the other as below.

```
Program Mixed_words;

Var
     signed_value        :integer;
     unsigned_value      :word;

Begin
     signed_value := 5ØØ;              {okay}
     unsigned_value := signed_value;   {error during PAS1}
End.
```

When compiling this program, the compiler will issue a warning to the effect that the types are not compatible. In order to use a mixture of INTEGER and WORD types we must use one of the simple functions in Pascal. This is the WRD function. It converts any INTEGER (or other ordinal value) into a WORD type value so that the usage will be compatible. Taking the example above, we could write the assignment statement as follows.

```
unsigned_value := Wrd (signed_value);
```

BYTE Type

Similar to the WORD type is the BYTE type. As you probably guessed, it occupies one byte in memory as shown in Figure 2-5. This type can be used in the same way the WORD type is used, except that the highest value for a byte is 255. The two types are not compatible without using the WRD function. Occasionally it is useful to declare a data element as a BYTE, but the compiler will allocate two bytes of memory for it anyway, because in Pascal, all data elements must be located at an even address in memory.

CHAR Type

This variable type is used to represent ASCII characters in a Pascal program. A CHAR value is any of the characters that can be typed on the keyboard. This includes all of the regular displayable characters as well as special control characters. They are different from byte variables because they can not be used in arithmetic operations.

One Character at a Time

The CHAR type is especially useful when dealing with characters one at a time. This could be a single character entered as a menu selection or as a response to some other prompt. The Likeit program shown in Figure 2-6 shows an example.

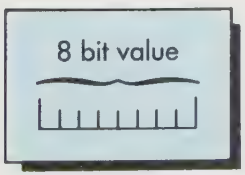


Figure 2-5. Internal representation of BYTE type

Figure 2-6. Example of CHARacter type

```
{*************************************************************
LIKEIT.PAS      Example of CHAR type use.
-------------------------------------------------------------
        This program asks you a loaded question.
-------------------------------------------------------------

Program Likeit (input,output);

Var
        answer      :char;

Begin
        Write ('Do you like Pascal? ');
        Readln (answer);

        If (answer = 'Y') or (answer = 'y')
                Then Writeln ('Great!  Lets go on')
                        Else Writeln ('Keep an open mind');
End.

-------------------------------------------------------------
Figure 2-6.      LIKEIT.PAS      Example of CHAR type
*************************************************************}
```

Note that character values within a program are written between two single quotation marks, or apostrophes. Lower case characters are different from upper case characters, so it is usually advisable for the program to test for either in order to be "user-friendly."

Alternate Characters

As you may have noticed, the IBM PC has a whole set of *alternate* characters. These produce special symbols on the monochrome display which can be used to dress up the display of information on the screen. These characters can be invoked from the keyboard by holding the Alt key down, entering the ASCII value for the character, and then releasing the Alt key. For example, a horizontal line can be displayed in a character position on the screen by using 196.

The CHR Function

To display these alternate characters directly from a Pascal program we need only use the simple function CHR. This is similar to the CHR$ function in BASIC. It takes any numeric value from 0 to 255 and treats it like a character. So, if you wanted to draw a horizontal line across the screen, you could use the Pascal program shown in Figure 2-7.

Figure 2-7. Example of CHR function

```
{*************************************************************
HORIZONT.PAS    Demonstrates the alternate character set.
-------------------------------------------------------------
        This program uses the alternate character set and the
CHR function to draw a horizontal line on the screen.
-------------------------------------------------------------
}
Program Horizont (output);

Const
        length = 8Ø;    { length of line }

Var
        hor_line        :char;
        count           :integer;
{
-------------------------------------------------------------
        Define the character and use a FOR...DO structure
to output a line of them on the screen.
-------------------------------------------------------------
}
Begin
        hor_line := Chr (196);  { <---- decimal value for char. }

        For count := 1 to length Do
                Write (hor_line); { display value 8Ø times }
End.
{
-------------------------------------------------------------
Figure 2-7.      HORIZONT.PAS      Example of CHR function
*************************************************************}
```

Since the variable "hor_line" is only one character, we need to use a *loop* to display it in each of the 80 character positions on the screen. Notice that we declared "length" as a CONSTant using the equal sign (=), since this is not the same thing as an assignment statement. Don't worry about the FOR...DO loop in this program. We will explore the whole family of looping statements in chapter 3.

BOOLEAN Type

Programming often involves determining whether or not something is true or false. This is usually in regard to a particular question which can be answered by "yes" or "no", such as:

Is the operator's password correct?

Does the data file exist?

Have all the data been entered?

These questions can also be represented as "true, false" statements as follows:

Password is correct.

File exists.

End of data reached.

It's Either FALSE or TRUE

In Pascal, a BOOLEAN type can have only the values FALSE or TRUE. The value of a BOOLEAN variable is generally interpreted during some conditional part of a program, such as an IF...THEN statement. Figure 2-8 illustrates the use of a BOOLEAN type.

Figure 2-8. Example of BOOLEAN type

```
{*****************************************************************}
BOOLEXAM.PAS    Example of BOOLEAN usage.
{-----------------------------------------------------------------}
       This program illustrates the use of a BOOLEAN type to
act as a "validation flag."  Here, two conditions are tested, and
the flag is set accordingly.
{-----------------------------------------------------------------}
{
Program Boolexam (input,output);
{
{-----------------------------------------------------------------}
                    Declarative division.
{-----------------------------------------------------------------}
{
Var
       number    :integer;
       flag      :boolean;
```

```
------------------------------------------------------------------
      Main program.  Input a number and check its range.
------------------------------------------------------------------
!
Begin
       Write ('Enter a number between Ø and 1ØØ ----> ');
       Readln (number);

------------------------------------------------------------------
       Start with the flag FALSE, then IF either condition is
true, set the flag TRUE.
------------------------------------------------------------------
!
       flag := false;

       If number < Ø
              Then flag := true;

       If number > 1ØØ
              Then flag := true;
!
------------------------------------------------------------------
       If flag is TRUE. display error message.
------------------------------------------------------------------
!
       If flag
              Then Writeln ('Range error');

End.
!
------------------------------------------------------------------
Figure 2-8.      BOOLEXAM.PAS      Example of BOOLEAN type
*****************************************************************!
```

Logical Conditions

As we shall see in chapter 3, the Boolean variables and the "false, true" values are at the heart of all the IF...THEN, WHILE...DO, and REPEAT...UNTIL structures. There are also several Pascal keywords that are Boolean by nature. For example, the ODD function returns a BOOLEAN value of TRUE if the function's argument is an odd number, and FALSE if it is an even number. The predeclared BOOLEAN variables EOF and EOLN are used in conjunction with files. EOF is TRUE if the end of a file has been reached during processing. EOLN is TRUE when the end of a line has been detected during input from the keyboard or a file.

BOOLEAN Operators

There are some special operators that apply to BOOLEAN types. If you remember your elementary logic, then these will be old friends. They are AND, OR, and NOT, and here's how they work:

TRUE AND TRUE = TRUE
TRUE AND FALSE = FALSE
FALSE AND FALSE = FALSE

TRUE OR TRUE = TRUE
TRUE OR FALSE = TRUE
FALSE OR FALSE = FALSE

NOT TRUE = FALSE
NOT FALSE = TRUE

These operators are not to be confused with the "bit logic operators" that go by the same names and are used for actually manipulating the bits in a BYTE or WORD type.

Enumerated Types

The BOOLEAN variables described above are one form of the *enumerated data type*. Enumeration means that the set of values which are permitted to be assigned to the variable are declared by the programmer. For BOOLEAN types, they are predeclared with the set of values (FALSE, TRUE).

Abstract Data Types

Other enumerated types can be declared with any kind of *abstract* meaning that happens to be useful in the application. For example:

```
Type
    day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
    time_of_day = (morning, noon, night);
    meal = (breakfast, lunch, dinner);
    vegie = (lettuce, carrot, onion, cucumber);
    fruit = (apple, orange, banana, pineapple);
```

Using abstract data types makes programs much easier to follow and frees us from worrying if the second day of the week is Monday or Tuesday.

Subrange Types

Declaring a variable as a *subrange* type permits the Pascal runtime debugging options to monitor the value of the variable, and stop execution of the program if the value is out of range. This is done automatically in the case of INTEGERS, where the subrange is predeclared as (-MAXINT..MAXINT).

Suppose that you wanted to use the result of an arithmetic expression to index some data in a table. The table will contain 100 entries, and the arithmetic required to produce the entry number is assumed to be complex, and in INTEGER mode. We want to be sure that the result is within the allowable entry number range. So, we will declare a subrange by indicating the limits separated by two periods.

```
Program Chk_entry (input, output);
Type
     entry_number = 1..1ØØ;    {Valid range}

Var
     x, y, z    :integer;
     result     :entry_number;    {type defined above}

Begin
     readln (x,y,z);
     result := 12 * x - y div z;{Range checked when running}
     writeln (result);
End.
```

Not much of a program, right? However, assume that something else would be done with the result after the calculation.

The main thing to notice in this program is the TYPE statement. Through its use, we can declare any data type in a way that has meaning for the application at hand. In this example, we declare a data type called "entry_number" and establish it as a subrange from 1 to 100. Then we declare a variable "result" as being of the "entry_number" type. When this variable is assigned the value obtained from evaluating the expression, a runtime error will occur if it is outside the specified subrange.

REAL Type

Thus far in our exploration of variable types we have only been able to deal with numeric values that are whole numbers within a limited range. In many computer applications, these restrictions get in the way. For example, how can we represent represent millions of dollars, or numbers with fractional parts? In Pascal, we use the REAL type. This type can take on a wide range of numeric values because of the way it is represented in memory. Remember, INTEGER and WORD types are represented as 16-bit values, so arithmetic operations on these variables can be performed directly by the processor using the internal 16-bit registers.

Scientific Notation

In order to deal with numbers whose binary representation would require more than 16 bits, Pascal uses a type of *scientific notation*. The way scientific notation works is that a value is expressed as a number multiplied by ten raised to some power. Any number can be expressed in this manner. Some examples:

$12{,}462 = 1.2462 * 10^4$

$16.5 = 1.65 * 10^1$

$.05 = 5.0 * 10^{-2}$

$-56.2 = -5.62 * 10^1$

Pascal uses "e" to indicate the exponent that follows, in this format:

(mantissa) e (exponent)

where the *mantissa* represents the actual numeric value, and the *exponent* represents the power of ten to which it is raised. Some more examples, then, using Pascal notation:

```
75,005          = 7.5005e4
-.00257         = -2.57e-3
29,650,299.50   = 2.96502995e7
10              = 1e1
```

Limitations of REAL Types

As you can see, there is quite a wide range of numbers that can be represented using REAL type variables. However, there are limits here as well. As with other types, these limits have to do with the way the variable

is represented in memory. In IBM Pascal, a REAL type variable occupies 32 bits of memory in 4 consecutive bytes, as shown in Figure 2-9. The first 3 bytes contain the 24-bit mantissa (including the sign), while the 4th byte contains an 8-bit binary exponent. Note that this is not the decimal exponent. Rather, it is the power to which 2 is raised in order to represent the decimal exponent. With this knowledge of the internal representation of REAL type variables we can determine their range limits.

The mantissa part will establish the actual numeric composition of the number, while the exponent part will determine the magnitude. The largest magnitude, positive or negative, is 2 to the power of 127. The maximum value for this number is approximately: 1.701412e38.

Arithmetic Operations

Since these REAL types occupy four bytes of memory, they can not be handled in arithmetic expressions directly by machine instructions. Instead, arithmetic operations on REAL type variables are performed by Pascal functions. Executing these functions requires many more machine instructions than are required for INTEGER types. As a result, programs which use REAL types will run slower than those which use INTEGER types.

All of the arithmetic operators may be used in expressions involving REAL types. The usual symbols "+", "−", "*", and "/"are used to indicate addition, subtraction, multiplication, and division. (Note that the slash "/" is used to indicate division of REAL types; the DIV function is used for INTEGER types. Figure 2-10 is a simple program that computes the area of a circle to illustrate the use of REAL type variables:

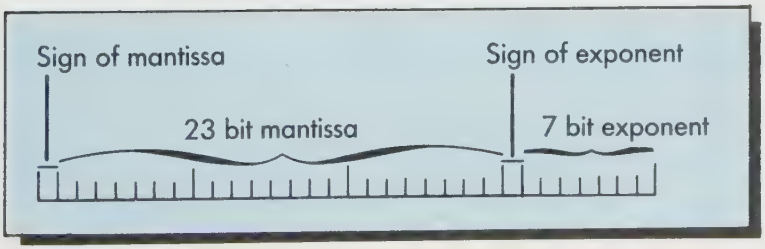


Figure 2-9. Internal representation of REAL type

Figure 2-10. Example of REAL type

```
{*****************************************************************
CIRCAREA.PAS    Calculate the area of a circle.
-----------------------------------------------------------------
        This program illustrates the use of REAL type data
elements to calculate the area of a circle, given its radius.
-----------------------------------------------------------------
}
Program Circarea (input,output);
{
-----------------------------------------------------------------
                    Declarative division.
-----------------------------------------------------------------
}
Const
        pi = 3.1416;

Var
        area,
        radius,
                        :real;
{
-----------------------------------------------------------------
               Main program.  Input the radius.
-----------------------------------------------------------------
}
Begin

        Write ('Enter the radius -----> ');
        Readln (radius);
{
-----------------------------------------------------------------
                Calculate and display the area.
-----------------------------------------------------------------
}
        area := pi * radius * radius;

        Writeln ('The area is ',area);

End.
{
-----------------------------------------------------------------
Figure 2-10.      CIRCAREA.PAS      Example of REAL type
*****************************************************************}
```

Notice the expression where the area is actually computed. In this case, the radius is simply multiplied by itself in order to effect the squaring. There is a REAL function that can be used to do the same thing. It is the SQR function, and the expression could be written as:

```
area := pi * Sqr (radius);
```

TRUNC, ROUND, and FLOAT Functions

Since there are so many data types in Rascal, it becomes necessary to convert them from one form to another in order to perform operations on them and still maintain compatibility. The following three functions are the most commonly used for conversion.

TRUNC (real_type)	returns INTEGER, dropping any fractional part
ROUND (real_type)	returns INTEGER rounded to the next highest whole number
FLOAT (integer_type)	returns REAL

There are several other functions which will be discussed throughout this book as they apply.

Figure 2-11 illustrates how the TRUNC function can be used to generate random numbers to simulate dice being rolled.

Figure 2-11. Example of TRUNC function

```
{*****************************************************************
DICE.PAS        Roll the Dice.
-----------------------------------------------------------------
        This program uses REAL types and the TRUNC function to
produce random values simulating the roll of two dice.
-----------------------------------------------------------------
}
Program Dice (input,output);

Const
        mult = 8.765432;          { any number between 1 and 100  }
        add = 0.534567;           { should be between .1 and .9   }

Var
        seed            :real;    { random number from 0 to 1     }

        die1,                     { two six-sided dice            }
        die2,
        total                     { total of the two dice         }
                        :integer;
```

```
        answer          :char;  { answer from user              }
                                                                  }
}
-------------------------------------------------------------------
        Roll the dice, and display the results, as long
as the player is willing.
-------------------------------------------------------------------
}
Begin
Write   ('Enter your lucky number ---> ')
readln  (seed);
        Repeat                  { until answer = 'n'

                seed := seed * mult + add;
                seed := seed - Trunc (seed);  { next seed       ;

                die1 := Trunc (6 * seed) + 1; { set first die

                seed := seed * mult + add;
                seed := seed - Trunc (seed);  { next seed

                die2 := Trunc (6 * seed) + 1; { set second die

                total := die1 + die2;   { total of both dice    ;

                Writeln (die1,die2,total);

                Write ('Roll again ? (y/n) ');
                Readln (answer);

        Until  (answer = 'n');  or  (answer = 'N');

end.
{
-------------------------------------------------------------------
Figure 2-11.      DICE.PAS      Example of TRUNC function
*******************************************************************}
```

This program works by using three assignments that produce a random die roll each time they are executed. The first is:

```
seed := seed * mult + add;
```

and is used to assure us that the value of "seed" will jump around a lot to give us the random effect. The second statement:

```
seed := seed - Trunc (seed);
```

changes "seed" to a value between zero and one by subtracting off the whole number portion. Finally, we get an integer from one to six with the expression:

```
die1 := Trunc (6*seed) + 1
```

Two dice are simulated and the results totaled. To change the sequence of rolls, enter a different lucky number.

Other REAL Functions

There are also several *intrinsic* functions available in IBM Pascal that provide a variety of operations. They can be used in a Pascal program without being declared. They include:

ABS(X)	the absolute value of X
SQR(X)	the square of X
SQRT(X)	the square root of X
SIN(X)	the trigonometric sine of X (where X is in radians)
COS(X)	the cosine
ARCTAN(X)	the arc-tangent
EXP(X)	exponential (e to the X power)
LN(X)	the natural logarithm of X

There are a host of other REAL functions which are not intrinsic, and must be declared as *external* in the Pascal program. Most of these deal with additional trigonometric functions; a few involve the use of two REAL arguments. They are amply explained in the IBM manual.

STRING and LSTRING Types

Technically, STRING and LSTRING are *structured* data types, since they are composed of more than one data element. In standard Pascal (as opposed to IBM Pascal), there is no such thing as a string. Instead, groups of characters that constitute a string must be declared as:

```
Var
    name      :packed array[1..25] of char;
```

The "packed" keyword means that there will be one character per byte. However, since IBM PC Pascal includes an extended data type called

super array, and the types STRING and LSTRING have been predeclared, we can shorten our variable declaration like this.

```
Var
     name      :string(25);  {maximum length of 25}
```

We'll explain more about super arrays in chapter 4. The important thing here is simply to become familiar with the form of the STRING type.

We have chosen to discuss STRINGs in this chapter because the concept is familiar to many BASIC programmers, and it will be useful to understand it at this point.

Constant and Variable STRINGs

We are used to thinking of a STRING as a group of characters that is treated as a unit. This might be a STRING constant employed as a prompt, or it might be a variable STRING that is entered into the program as data. Figure 2-12 contains an example of both of these STRING types.

Figure 2-12. Example of STRING type

```
{*****************************************************************
FRIENDLY.PAS    Example of STRING use.
-----------------------------------------------------------------
        This friendly little program uses a STRING to handle
groups of characters.
-----------------------------------------------------------------
}
Program Friendly (input,output);

Const
        prompt = 'Hey baby, whats your name';

Var
        name,       :string(25);
{
-----------------------------------------------------------------
}
Begin
        Writeln (prompt);
```

```
        Readln (name);
        Writeln ('Have a nice day ', name);
End.
```

Figure 2-12. FRIENDLY.PAS Example of STRING type

Length of STRINGs

In IBM PC Pascal, there is a special type of STRING, called an LSTRING type. Basically, it is a STRING with its length attached to the front end. LSTRINGs are predeclared as are STRINGs.

```
Type
     lstring = super packed array[Ø..n] of char
Var
     name  :lstring(8Ø);  {maximum length is 8Ø}
```

The current length of the LSTRING is stored in its first character, name[0]. This length will only occupy one byte of memory, and therefore limits the length of an LSTRING type to 255 characters. The length of an LSTRING type can also be determined by using the notation:

```
name.LEN
```

the result of this is a BYTE type variable containing the character count. The recipient variable should therefore be of type BYTE, to be compatible. We will be using several examples of the .LEN notation in chapter 4, where we will show more examples that use both STRING and LSTRING.

One commonly used function for LSTRINGs is CONCAT which combines two strings into one long one.

```
Var
    first_name, last_name :lstring(2Ø);
    full_name  :lstring(4Ø);
Begin
    last_name  :=  concat (last_name,',');
    full_name  :=  concat (last_name, first_name);
```

The two assignments above append a comma to the end of the last name, then combine the names in the order they might appear in the telephone book.

Summary

This concludes our discussion of Pascal's simple data types, expressions and functions. These provide the building blocks for most programs. We will see their use throughout the book, in every programming example. If you're not sure you've got it all, maybe a quick review is in order. Table 2-1 provides a quick overview.

Table 2-1. Simple data types

Simple Data Types

Type	Examples	Range of Values
INTEGER	1200 −33	−MAXINT...MAXINT (32767...32767)
WORD	33000 54321	0...MAXWORD (0...65535)
BYTE	65	0...255
CHAR	'A'	CHR(0)...CHR(255)
BOOLEAN	true	true,false
Enumerated	(red, blue, green) (Sun, Mon, Tue, Wed, Thu, Fri, Sat)	programmer's choice
Subrange	1...10 Mon...Fri	Any portion of the above ranges
REAL	3.14159 −0.003	+/−1.701412e38
STRING	'George Washington'	up to 32766 characters
LSTRING		up to 255 characters

Don't worry too much if every detail isn't clear. Press on to the next chapter, because there will be many more examples of these simple data types throughout the book.

Exercises

1. What are variables?

2. What is the difference between INTEGER and REAL variables?

3. How many Boolean values are there? How many CHARacters values?

4. What would be the result of the following expressions:
   a) 7 + 3 * 2
   b) (7 div 4) – 1
   c) Round (8.0 / 3.0)
   d) 7.3 – Trunc (7.3)
   e) (3 = 5)

Solutions

1. Variables are the data elements used by a program. They represent a portion of memory set aside to hold various values while the program is running.

2. INTEGER variables are only 2 bytes (16 bits) long and execute much faster than REAL variables, but they have a limited range and can not contain a fraction. REAL variables are 4 bytes (32 bits) long and require more time for arithmetic than INTEGER variables, but they cover a much larger range of values, including numbers with fractional parts.

3. BOOLEAN: 2 values (true, false)
   CHAR: 256 values (CHR(0)..CHR(255))

4. a) 13
   b) 0
   c) 3
   d) 0.3
   e) false

3

Program Control

Concepts

- Decision-making
- Conditional Programming
- Relational Operators
- Error Flags
- Compound Statements
- Iteration and Loops
- Repetitive Operations
- Escape Clauses

Keywords

IF...THEN...ELSE, CASE, OTHERWISE, FOR...DO, WHILE...DO, REPEAT...UNTIL, GOTO, BREAK, CYCLE, RETURN

We have learned how Pascal distinguishes among many different types of data. There seems to be a Pascal data type for just about any kind of application you can imagine. But what good is all this data capability if there isn't a way to use it, interactively? A program that says "Hey baby..." every time it executes (as in the previous chapter) is really nothing more than a glorified answering machine, regurgitating its message no matter who calls. What we need is an answering machine that can **modify** its message depending upon who's calling. Then we would have an "intelligent machine," one that makes decisions.

Conditional Programming

The real key to the computer's decision-making power is that no matter how complex the facts might be, they can all be broken down into

a number of simple decisions. These decisions are so simple that they can be dealt with as "black and white," "yes or no," "true or false" issues. There will never be the "shade of grey" which we as humans must contend with.

"To be, or not to be — that is the question." In a nutshell, that's just how the computer views the world. Either a condition exists, or it does not, and decisions can be made upon that premise. In the following sections, we will discuss how the computer analyzes the facts given to it.

The IF...THEN Statement

In almost every programming language the primary mechanism for dealing with decisions is the IF...THEN statement. Simply stated, it indicates that

IF (some condition exists) THEN (do something)

Of course, there are a few variations, but that's the main theme. If the condition exists, then the program will do the "something" following the THEN keyword. If it does not exist, then the "something" will not be done. The statement always works the same way no matter how many times the program is executed.

In Pascal, we can add another phrase to the statement.

IF (some condition exists)

THEN (do something)

ELSE (do some other thing)

This scheme looks like a fork in the road. But the program will never suffer the indecision Robert Frost had with "The Road Not Taken." This is because the choice is implicit in the conditions that lead to the fork. The program can only take one path: the correct one. (That is, assuming the program has been written correctly.)

The Condition Clause

There are many ways of expressing the *condition* that will determine the outcome of the IF...THEN statement. The most common is to state some mathematic relationship and make the decision based upon whether the relationship is true or false. An example we can all relate to is:

IF (checking balance equals zero)

THEN (stop writing checks)

ELSE (live it up)

The expression that directly follows the IF part of the statement is called the *condition clause*. It may take several forms, but will always be evaluated as a Boolean expression; that is, the condition clause will always evaluate to TRUE or FALSE.

Evaluating an arithmetic relationship as the condition clause involves the use of relational operators.

=	equality
<	less than
>	greater than
<=	less than or equal
>=	greater than or equal
<>	not equal

Here are some examples to give a general idea:

check_bal = 0	TRUE if zero, FALSE if positive or negative
check_bal < 0	TRUE if negative, FALSE if zero or positive
check_bal <= 100.00	TRUE if 100.00 or less, FALSE if over 100.00.

The arithmetic appearing in the condition clause may be quite complex, involving different data types (as long as compatibility rules are followed), arithmetic operators, and functions. The main thing to remember is that the condition clause must eventually boil down to a Boolean TRUE or FALSE. Here are some more complex condition clauses:

```
SQRT (SQR (side_a) + SQR (side_b)) < min_hypot
height * length * width > max_volume
SQR (SQRT (number)) <> number
(a + b + c) = (d + e + f)
```

The foregoing examples all had to do with numeric data. We can also use CHARacter type data in the condition clause of an IF statement. This is often useful when interpreting the answers to prompts such as:

```
Write ('Continue Y/N ');
Read (answer);
IF answer = 'Y' THEN...
```

Something rather interesting about this kind of decision-making is that often the decision can be approached from either side of TRUE or FALSE. For example, the previous code segment would produce similar results if written:

```
Write ('Continue Y/N ');
Read (answer);
IF answer <> 'N' THEN...
```

Here, instead of looking for the answer to be equal to "Y," we are checking to see that it is not "N." This approach, while functional, leaves some loopholes for errors. It assumes that since the answer is not "N" that a "Y" response is implied. The former approach demands an affirmative response; it is generally safer since it does not make assumptions. We'll discuss this again later in this chapter.

Using a Flag for Validation

Since the condition clause of the IF statement breaks down to a Boolean TRUE or FALSE value during execution, it is possible to use a BOOLEAN type data element as the condition clause, rather than a relational expression. This is not usually necessary unless there are several factors influencing the condition at different times during program execution. In such cases, a flag may be used as shown in Figure 3-1.

Figure 3-1. Example of IF...THEN statement

```
{*****************************************************************
VALIDATE.PAS    Using an error flag.
-----------------------------------------------------------------
        This program demonstrates how an error flag may be
set as a result of a variety of validation testing.
-----------------------------------------------------------------
}
Program Validate (input,output);

Var
        a, b, c, d      :integer;
        error_flag      :boolean;
{
---------------------------------------+-------------------------
}
Begin
        error_flag := false;
        Writeln ('Enter 4 positive integers');
        Readln (a, b, c, d);
```

```
{
-------------------------------------------------------------------
       Test each integer and set the flag if any one is
less than or equal to zero.
-------------------------------------------------------------------

       If a <= 0 then error_flag := true;
       If b <= 0 then error_flag := true;
       If c <= 0 then error_flag := true;
       If d <= 0 then error_flag := true;
}
-------------------------------------------------------------------
          Now, use the flag as the condition clause.
-------------------------------------------------------------------
{
       If error_flag then Writeln ('Error in data');

End.

-------------------------------------------------------------------
Figure 3-1.    VALIDATE.PAS      Example of IF...THEN statement
*******************************************************************}
```

In this example, we initialize the BOOLEAN type variable "error_flag" with the value FALSE, since no error exists at the beginning of the program. Then, the program asks that four positive integers be entered. In a series of four IF...THEN statements, each of the four values input is checked to make sure it is positive.

If any one of the variables is less than or equal to (<=) zero then "error flag" is assigned a TRUE value. Finally, "error_flag" itself is used as the condition clause of an IF...THEN statement. A TRUE value will cause the WRITELN following the THEN to be executed and produce an error message. This is called the "true statement." A value of FALSE would result in the program ending without the error message being printed.

The False Statement

There is one more option to the IF statement: the "false statement." This is the statement that will be executed if the result of evaluating the condition clause is FALSE. In the last example, the FALSE statement was implied to end the program. This is called *falling through* an IF statement, since there is no specified statement to be executed if the result is FALSE.

We specify a false statement by adding the Pascal keyword ELSE followed by the false statement. For the previous example, we could change that last IF statement as follows:

```
If error_flag
     Then Writeln ('Error in data')
     Else Writeln ('Thanks for doing it right');
```

Variations

Have you ever heard the expression "there's more than one way to skin a cat"? As a cat lover, I have always somewhat resented it, but the idea is still valid. IF there ever was a case when this was true, THEN it applies to the IF..THEN statement! There are so many permutations of this statement that if 10 programmers had been commissioned to write the example program above, there would be 10 different approaches used.

Pascal allows *compound expressions* to be used as the condition clause. Each part of the compound expression is evaluated for TRUE or FALSE and the rules of logic described earlier are used to yield the compound result. To accomplish this, the Boolean operators AND, OR, NOT are used. With this approach, we can perform the value testing in the above example as follows:

```
If (a <= 0)  or
   (b <= 0)  or
   (c <= 0)  or
   (d <= 0)
     Then Writeln ('Error in data')
     Else Writeln ('Thanks for doing it right');
```

Here, we have composed one IF statement that checks all four of the input variables, one at a time. The resulting TRUE or FALSE values from each test are logically tied by ORs to obtain the final value of the condition clause. Remember how the logical OR works? If any argument is TRUE, then the entire expression evaluates TRUE.

Notice that each relational test is enclosed within parentheses. This is to make sure that the compiler understands what we want. To illustrate, let's just write part of the statement a little differently.

```
If a <= 0 or b <= 0
     Then...
```

Without the parentheses, the compiler would interpret the statement as:

```
If a <  (Ø or b) <= Ø
    Then...
```

This is because the arguments for the OR operation are assumed to be the immediately adjacent expression terms, one on each side of the word OR. Of course, this assumption makes the rest of the condition clause nonsensical, and the compiler would flag the statement as an error.

Both Sides of the Fence

Let's look at an interesting phenomenon surrounding the IF statement. Remember that we can approach a decision from either the TRUE side or the FALSE side. In the example above, we have been looking at the FALSE side. That is, the four values that are input are supposed to all be positive integers. We are checking to see if any one of them is *not* positive, and making the decision on that basis. We can also use the TRUE approach, and check to see that each value is in fact positive.

```
If (a > Ø)  and
   (b > Ø)  and
   (c > Ø)  and
   (d > Ø)
     Then Writeln('Thanks for doing it right')
     Else Writeln('Error in data');
```

Notice how this approach turned the whole statement inside out. The relational operator is now ">" (greater than); the terms in the condition clause are tied by ANDs instead of ORs, and the two messages have exchanged places. Remember how the logical AND works? If any one of the terms is FALSE then the entire condition clause is FALSE.

Compound TRUE and FALSE Statements

The TRUE and FALSE statement parts of the IF statement can be compound just as the conditional part can. That means that several program statements can be inserted as the TRUE or FALSE statement.

There are two methods for accomplishing this in IBM Pascal. One way is to group the elements of the compound statement using brackets [].

```
If (condition clause)
     Then [                          {begin true}
          (true statement #1);
          (true statement #2);
            *       *       *
            *       *       *
          ]                          {end true}

     Else [                          {begin false}
          (false statement #1);
          (false statement #2);
            *       *       *
            *       *       *
          ];                         {end false}
```

The second method, using BEGIN and END, is more traditional Pascal.

```
If (condition clause)
     Then begin
          (true statement #1);
          (true statement #2);
          end

     Else begin
          (false statement #1);
          (false statement #2);
          end;
```

Nested IF...THEN Statements

Since we can hang any group of Pascal statements on the true or false "hook" of a condition clause, why not include another IF statement there as well? Here's where it starts to get complicated, and you must be very clear about your program design before you begin. This technique is called the *nested* IF statement, and it looks like this:

```
If (condition clause #1)
     then If (condition clause #2)
               then (true statement #2)
               else (false statement #2)
     else If (condition clause #3)
               then (true statement #3)
               else (false statement #3);
```

This example uses only two levels of nesting. You can see that without too much more code, an entire program could be written using one IF...THEN...ELSE statement. Figure 3-2 illustrates the use of nested IF...THEN statements in an updated version of the old "guess the number" game. You pick a number and then press Enter to start the program guessing. It uses the binary search technique to home in on the number. Depending upon how you answer the "high or low" question (be honest), the program will shrink the guessing range, and guess again. Notice the nested IF...THEN statement that determines which way to adjust the range. (We'll explain the REPEAT...UNTIL structure later in this chapter.)

Figure 3-2. Example of nested IF statements

```
{*****************************************************************
GUESSER.PAS     Program guesses the number.
-----------------------------------------------------------------
        Choose a number.  The program will make a guess.  Tell
the program if its guess was high or low, and it will guess again
until it finds the correct number.
-----------------------------------------------------------------
}
Program Guesser (input,output);

Const
        highest = 1000;          { highest possible guess        }

Type
        gssrng = 1..highest;     { range of possible guesses     }

Var
        high,
        low,                     { limits after questions        }
        guess                    { guess of number               }
                :gssrng;

        answer  :char;           { answer to questions           }
{
-----------------------------------------------------------------
         Main program.  Prompt and start with ENTER key.
-----------------------------------------------------------------
}
Begin

        Writeln ('Pick a number between 1 and', highest );
        Write ('Press Enter when ready');
        Readln;
```

```
        low := 1;
        high := highest;        { starting range                }
{
-------------------------------------------------------------------
        Stay inside the REPEAT...UNTIL structure until the
number is guessed correctly.
-------------------------------------------------------------------
}
        Repeat { until number is guessed }

            guess := low + (high - low) Div 2; { middle of range }

            Repeat
                Write ('Is ', guess, ' high, low, or correct? ');
                Readln (answer);
            Until (answer = 'h')  or
                        (answer = 'l')  or
                        (answer = 'c');
{
-------------------------------------------------------------------
       Use this "nested IF" statement to adjust the range.
-------------------------------------------------------------------
}
            If answer <> 'c'
                Then If answer = 'h'
                        Then high := guess - 1 { too high       }
                        Else low := guess + 1; { too low        }

        Until answer = 'c';     { keep guessing until right     }

End. { guesser }
{
-------------------------------------------------------------------
Figure 3-2.      GUESSER.PAS      Example of nested IF statements
*******************************************************************
```

As you continue to learn Pascal, and write your own programs, you will notice the inevitable use of the IF statement. There are hardly any applications of computer programming that do not involve decision-making of this nature.

The IF concept gives a program the semblance of intelligence; with it, a program appears to be aware of events that occur within its realm and able to change its behavior accordingly.

The CASE Statement

The CASE statement is another quite common statement in programming languages. In BASIC, it is known as ON...GOTO. In Fortran, it is called an "indexed GO TO." It has a nicely structured form in Pascal, but is basically the same as other conditional programming concepts. The CASE statement is like an IF statement that can have more than the two paths TRUE and FALSE. In fact, you can build as many paths as you need. Here is the general form of the CASE statement.

```
Case (expression) of
     (index #1):    (statement #1);
     (index #2):    (statement #2);
     (index #3):    (statement #3);
          *              *
          *              *
          *              *
     (index #n):    (statement #n);

          Otherwise
               (exception statements);
end;
```

The CASE Index and Constants

When the CASE statement is executed, the case "expression" will be evaluated to become the case "index." This index value is then compared to each of the index values listed. These are called the case constants. Each case constant is associated with a statement (notice the colon ":" between them). The statement will be executed only if the current value of the case index is equal to its case constant. This one-to-one relationship is essential to the correct operation of the program structure. Obviously, we can't have the same case constant associated with two different statements. The Pascal compiler will detect this as an error.

The CASE Exception

If the evaluation of the case index yields a value that is not in the list of case constants, the "exception statements" following the OTHERWISE keyword are executed. This is an optional part of the CASE statement and may be omitted without causing a compile error. However, a stray case index will cause an error when the program is run. It's always good programming technique to provide some path to take when the anticipated structure fails.

Example of a CASE Statement

The CASE statement provides an efficient way to program a menu of processes. Each process is associated with a case constant, so that the value of the case index will determine which of the them will be executed. Figure 3-3 illustrates how the CASE statement can be used as a menu.

Figure 3-3. The CASE statement as a menu

```
{*************************************************************
CASEMENU.PAS    Example of CASE Statement as a Menu.
-------------------------------------------------------------
        This program uses a CASE statement as the primary
structure in driving a menu program.
-------------------------------------------------------------

Program Casemenu (input,output);

Var
        filename,
        order_no,
        report_date,
        util_name,
                        :string(8);

        answer          :char;
{
-------------------------------------------------------------
      Main program.  Display the menu, and input selection.
-------------------------------------------------------------
}
Begin
    Repeat
        Writeln;
        Writeln ('***** Main System Menu *****');
        Writeln;
        Writeln ('File Maintenance .....   f');
        Writeln ('Order Entry ..........   o');
        Writeln ('Sales Analysis .......   s');
        Writeln ('Utility Programs .....   u');
        Writeln;
        Write   ('Enter selection -------> ');

        Readln (answer);
```

```
{
-------------------------------------------------------------------
    Use the CASE structure to perform conditional statements.
-------------------------------------------------------------------
}
        Case answer of
                'f':    [
                        Write ('Enter file name -------> ');
                        Readln (filename);
                        ];
                'o':    [
                        Write ('Enter order number ----> ');
                        Readln (order_no);
                        ];
                's':    [
                        Write ('Enter report date -----> ');
                        Readln (report_date);
                        ];
                'u':    [
                        Write ('Which utility? --------> ');
                        Readln (util_name);
                        ];
{
-------------------------------------------------------------------
                    The exception clause.
-------------------------------------------------------------------
}
                Otherwise
                        Writeln ('Invalid selection');
        end;  {case}
{
-------------------------------------------------------------------
                        Continue option.
-------------------------------------------------------------------
}
        Write ('Another selection?  (y/n) ');
        Readln (answer);

    Until answer = 'n';
End.
{
-------------------------------------------------------------------
Figure 3-3.      CASEMENU.PAS      The CASE statement as a menu
*******************************************************************'
```

The menu itself is simply contained within the literal constants in the WRITELN statements. A single CHAR type is input as the answer to the menu selection.

Each CASE index is associated with a compound Pascal statement. In our example, we just "fake" actually doing different things for different selections. However, as we shall see in chapter 5, "Functions and Procedures," we could add a procedure call within each compound statement to accomplish something useful.

Notice the OTHERWISE clause. It will cause an error message to be displayed if an invalid menu selection is made.

Enumerated and Subrange CASEs

IBM Pascal also allows a range of values or a group of values to be used as case constants. We can use this capability to make the previous example a little more intelligent by recognizing either upper or lower case characters.

```
Case option of
     'F','f':       { file maintenance }
     'O','o':       { order entry }
     'S','s':       { sales analysis }
     'U','u':       { utility programs }

         Otherwise ...
```

Here we have two case constants, separated by commas, associated with each executable statement. Either one will invoke the statement.

Use of a *range* as the case constant provides a means of associating data by category. In Figure 3-4, we will input test scores and accumulate them by subrange to establish the distribution curve.

Figure 3-4. A subrange CASE statement

```
{*****************************************************************
SCORDIST.PAS    Score Distribution Using CASE statement.
-----------------------------------------------------------------
       This program illustrates the use of a subrange as the
CASE statement index.
-----------------------------------------------------------------
}
Program Scordist (input,output);

Var
       score,              { score for each student }
       grade_A,            { count of scores for each grade }
       grade_B,
```

```
        grade_C,
        grade_D,
        grade_F
                        :integer;

Value
        grade_A := Ø;       { start all counts all zero }
        grade_B := Ø;
        grade_C := Ø;
        grade_D := Ø;
        grade_F := Ø;
;
-------------------------------------------------------------------
        Main program.  Input scores until a negative score
is entered.  Use the CASE statement to accumulate the score
distribution.
-------------------------------------------------------------------
}
Begin

        Writeln ('Enter test scores, negative to end');
        Write ('Score ----> ');
        Readln (score);

-------------------------------------------------------------------
            Use test score subranges for CASE index.
-------------------------------------------------------------------

        While score >= Ø Do
                {
                Case score of
                        Ø..59:          grade_F := grade_F + 1;
                        6Ø..69:         grade_D := grade_D + 1;
                        7Ø..79:         grade_C := grade_C + 1;
                        8Ø..89:         grade_B := grade_B + 1;
                        9Ø..1ØØ:        grade_A := grade_A + 1;
                        Otherwise
                                end;

                Write ('Score ----> '); { input the next score  }
                Readln (score);
                }:

-------------------------------------------------------------------
              Now, output the score distribution.
-------------------------------------------------------------------

        Writeln;
        Writeln ('Score Distribution');
        Writeln ('A', grade_A);
```

```
        Writeln ('B', grade_B);
        Writeln ('C', grade_C);
        Writeln ('D', grade_D);
        Writeln ('F', grade_F);

End.
```

Figure 3-4. SCOREDIST.PAS A subrange CASE statement

We will use the INTEGER variable "score" to be the case index. Then, each of the case constants is really an integer subrange. Each time a score is entered, it will filter through the CASE structure, causing one of the statements to be executed. (We'll be explaining the WHILE...DO statement later in this chapter.)

If "score" should contain a value other than those represented as case constants, the OTHERWISE clause will be executed. This simply ends the case structure, without executing any of its statements. Thus, a score of less than 0 or greater than 100 will be ignored.

Iteration Control Statements

In the previous examples, we used some new Pascal keywords: REPEAT...UNTIL, and WHILE...DO, without explaining how they worked. These are two of the methods Pascal offers to do something in a repetitive manner, or to perform *iteration*. The need for iteration is very common in programming. Rarely will we write a program that executes from top to bottom without repeating something. Many times the iteration consists of performing the same operations on multiple data elements, as in the SCORDIST program. At other times, iteration may also involve recursion. This means that some data is processed, and then the result of that process is processed again, and so on.

While the concept of iteration in programming is quite common, Pascal does it with a great deal of style. There are three different approaches to iteration control. They are so simple that they are practically self-explanatory. They are:

FOR...DO

WHILE...DO

REPEAT...UNTIL

We'll cover each of these statements in the following sections.

The FOR...DO Statement

The FOR...DO statement is very similar to the FOR...NEXT loop in BASIC. It provides a way of executing a section of program a specified number of times. To achieve this, the program must count each time it does the process, and also check that it does it the correct number of times. The general form of the FOR...DO statement is as follows.

```
FOR control_var := initial_val TO final_val
     DO statement;
```

Control Variable, Initial Value, and Final Value

The *control variable* is what does the counting in the FOR statement. The first time the FOR statement is encountered, the control variable will be set to the initial value. Then the statement following the DO is executed, as long as the final value has not been reached. If it has, the FOR statement is finished executing, and the program proceeds to the next statement. If the final value has not been reached, the control variable is *incremented* automatically, and the statement part is executed again. Here are some simple FOR...DO examples.

```
For number := 1 to 1ØØ
     Do Writeln (number);

For position := 1 to 8Ø
     Do Write ('*');

For power := first to last
     Do number := number * 2;
```

The first example simply prints out the value of the control variable "number" each time through the loop. The second example prints 80 asterisks on the screen. In the third example, the variable "number" is doubled successively each time through the loop.

An interesting thing to note about the last example is that the statement part will not be executed at all if the current value of "last" happens to be less than the current value of "first." This feature adds a bit of the IF statement condition testing ability to the FOR statement.

There are some rules for using the FOR statement. The control variable and the initial value and final value must be type-compatible. Also, the control variable must not be changed by any statement within the repetitive loop. After the termination of the FOR statement, the control variable might be any value, but it may be assigned a new value or used in another FOR statement. The control variable cannot be passed

as a reference parameter to a function or procedure (see chapter 5, Functions and Procedures).

The Reverse Option: DOWNTO

There is a *reverse option* to the FOR...DO statement. Instead of starting low, and increasing the control variable by steps, or *incrementing* it, to some final value, you can start high, and decrease by steps, or *decrement* it, to some final value. This is like a "count-down" and can be very useful. Here is the general form.

```
FOR control_var := initial_val DOWNTO final_val
     DO statement;
```

Here again, the statement will not be executed at all if the "initial value" starts out lower than the "final value." Any ordinal type can be used as the control variable, so it is also possible to use characters.

Example of FOR...DO Statement

Figure 3-5 is a one-statement program to output every ASCII character to the monochrome display.

Figure 3-5. Example of FOR...DO statement

```
{*************************************************************
ASCIIOUT.PAS    Output full ASCII set using FOR...DO.
-------------------------------------------------------------
       This program uses a FOR...DO statement to output the
full 256 ASCII character set to the display.
-------------------------------------------------------------
}
Program Asciiout (output);

Var
       decimal   :integer;
{
-------------------------------------------------------------
}
Begin
       For decimal := Ø to 255 Do
              Write (Chr (decimal));
End.
{
-------------------------------------------------------------
Figure 3-5.      ASCIIOUT.PAS      Example of FOR...DO statement
*************************************************************}
```

Now, let's carry this one step further. Remember how the IF statement could have compound statements in its TRUE and FALSE clauses? Well, this feature is also available in the FOR...DO statement. Once again, there are two methods that can be used.

[ ]	the bracket method
begin... end;	pseudo procedure method

One very useful application of the FOR...DO statement is in processing components of a structured data type. We will treat this whole area of component processing, or *indexing*, in chapter 4. For now, let's move on to the second of Pascal's repetitive statements.

The WHILE...DO Statement

The WHILE...DO statement is practically self-explanatory. WHILE some condition exists, a statement, or group of statements following the DO, are executed. It's like the FOR...DO statement, in that it can loop anywhere from zero to forever times. However, the WHILE statement does not use an index variable the way that the FOR statement does. Instead, its operation is dependent upon the Boolean expression that composes the condition clause for the WHILE...DO statement. The general form of the statement is:

```
While (condition clause)
     Do (TRUE statement);
```

The Condition Clause

The rules for the condition clause in the WHILE...DO statement are just like those for the IF..THEN statement. A Boolean expression is used to establish the condition being tested, and the result will be either TRUE or FALSE. As long as the evaluation of the condition clause is TRUE, the statements following the DO will be executed.

The condition clause is evaluated **before** the statements are executed. Then before executing them again, the condition clause is re-evaluated. If the result is FALSE, the statements following the DO are skipped, and execution resumes at the next statement. Notice that the condition clause is evaluated the very first thing when the WHILE statement is executed. If the result of the first evaluation is FALSE then the statements following the DO will *never be executed*.

Here are some simple examples of the WHILE...DO statement.

```
While check_amt <= balance
     Do Write_check;

While intensity < max
     Do intensity := 1.44 * intensity;

While not error_flag
     Do   |
          ... process ...
          ];
```

Here again, we prefer the use of the brackets "[" and "]" to the words BEGIN and END, but either may be used to group statements together in the statement clause of the WHILE.

In the last example we also see the Boolean operator NOT. This causes the Boolean result of evaluating the condition clause to be inverted. Therefore, execution of the statements following the DO will continue as long as the condition is FALSE. This is because NOT FALSE = TRUE, and so the statements are executed. Only when the actual condition becomes TRUE will the statements following the DO be skipped. Later we will see how this technique is used for looking-up items in a table, and processing structured data types.

Examples of WHILE...DO Statement

Figure 3-6 is an example using the WHILE statement to control data entry. The program loops continuously, reading numbers and outputting their square roots until the user enters zero.

Figure 3-6. Example of WHILE...DO data entry

```
{*****************************************************************
ROOTS.PAS       Computes the square root of numbers.
-----------------------------------------------------------------
        This program illustrates the use of the WHILE...DO
statement for repetitive data entry.
-----------------------------------------------------------------
}
Program Roots (input,output);

Var
        number,
        square_root     :real;
```

```
{-------------------------------------------------------------------
        Main program.  Input positive numbers and use the SQRT
function to produce their square root.
-------------------------------------------------------------------}

Begin

        Write ('Enter a number ----> ');
        Readln (number);

        While number > Ø Do
                {
                square_root := sqrt (number);
                Writeln ('The root is ', square_root);

                Write ('Enter a number ----> ');
                Readln (number);
                };

End.
{-------------------------------------------------------------------
Figure 3-6.      ROOTS.PAS      Example of WHILE...DO data entry
*******************************************************************}
```

WHILE statements can be nested the way IF statements can, and can also contain or be connected with IF statements. The program set forth in Figure 3-7 determines all of the prime numbers up to 10,000. (A prime number is one that is evenly divisible only by itself and by the number one.) The program uses a method called the "Sieve of Eratosthenes," which is really quite clever. Here you'll see an example of the WHILE statement within an IF, within a FOR statement.

The program starts testing numbers beginning with the number two, which is prime by definition. It contains a structured data type in the form of the variable "prime." This is an ARRAY OF BOOLEAN types. We'll cover arrays in the next chapter; for now, just think of an array as a list of flags similar to the flag that we used for data validation earlier in this chapter.

In the beginning of the program, we use a FOR...DO statement to "initialize" all of the flags in the array to a TRUE value. Then, the program enters another FOR...DO statement that "sieves out" the non-prime numbers. This is accomplished by setting the flag for all multiples of a prime number to the value FALSE. As each prime number is determined, it is displayed on the screen.

This program is an amusing one to type in and run, but the important point is to notice how a WHILE...DO statement can be nested in an IF...THEN statement.

Figure 3-7. Example of WHILE...DO statement

```
{*****************************************************************
SIEVE.PAS       Prime number sieve program.
-----------------------------------------------------------------
        This program uses the "sieve method" for determining
all the prime numbers up to "maxtest."  An array of BOOLEAN types
is used to flag the prime numbers.
-----------------------------------------------------------------
}
Program Sieve (output);

Const
        maxtest = 10000;        { highest number to test        }

Type
        index = 2..maxtest;     { range of number being tested  }

Var
        prime           :array [index] of boolean;
                                { set false when index is a
                                multiple of a prime             }

        test            :index; { value to test if prime        }

        i               :integer; { index for setting prime[]   }
{
-----------------------------------------------------------------
        Main program.  First initialize the array to TRUE.
Then start testing with the first element in the array.
-----------------------------------------------------------------
}
Begin

        For i := 2 to maxtest Do { initialize array             }
                prime[i] := true;
{
-----------------------------------------------------------------
      Look for a prime, set all multiples of it to FALSE.
-----------------------------------------------------------------
}
        For test := 2 to maxtest Do { look for primes           }
            If prime[test]      { true if "test" is prime }
                Then    [
                        Write (test);   { display the prime.    }
```

```
                    i := 2 * test;
                    While i <= maxtest Do
                            [       { set multiples false }
                            prime[i] := false;
                            i := i + test;
                            ];
                    ];

        Writeln;                    { end last line                 }

End.
```

Figure 3-7. SIEVE.PAS Example of WHILE, IF and FOR...DO

****

The REPEAT...UNTIL Statement

The REPEAT statement rounds out the Pascal trio of iteration control statements. Here is the general form.

```
Repeat
     (one or more statements);
Until (condition clause);
```

This statement is very much like the WHILE statement, except for one distinct difference. In the WHILE statement the condition clause is evaluated *before* the statements are executed, making it possible to bypass them entirely if the condition clause is FALSE the first time through. In the REPEAT statement, the condition clause is not evaluated until after the statements in its body have been executed. This means that the body will be executed *at least once* before the condition clause is evaluated.

Another thing to notice is the *inverted logic* between the WHILE and the REPEAT statements. The WHILE statement will execute its body *as long as* the condition is TRUE. The REPEAT statement will execute its body *until* the condition becomes TRUE.

Both the WHILE and REPEAT statements are useful in deciding when a process is finished. This is generally done by evaluating some data that is used in the program. For example, Figure 3-8 shows a program that inputs the names of people who will be players in a game. Not knowing how many people will be playing, we will allow entry of players' names until the word "end" is entered.

In this example we see the use of three forms of conditional programming. The REPEAT statement will cause the whole program to be repeated until there is a at least one player. The inner WHILE

statement will loop on the same entire part of the program as long as the name is not "End." And we threw in our old IF statement to display a [illegible] message if nobody wants to play.

The FOR...DO, WHILE...DO and REPEAT...UNTIL statements offer a wide choice in the way an iterative process can be carried out. With such a choice of logical structures, almost any set of conditions can be represented easily in Pascal.

Figure 3-8. Example of REPEAT...UNTIL statement

```
-----------------------------------------------------------------
[illegible]
-----------------------------------------------------------------
     [illegible]
[illegible]
-----------------------------------------------------------------

[illegible]

Const
     [illegible]                    [illegible]

Type
     [illegible]

Var
     [illegible]          [illegible]
     [illegible]          [illegible]

     [illegible]          [illegible]

[illegible]
     [illegible]          [illegible]

-----------------------------------------------------------------
     [illegible]
-----------------------------------------------------------------

begin
     [illegible]

     repeat
          [illegible]
          [illegible]
          [illegible]
          [illegible]
```

Escape Clauses

We may use either numeric labels like BASIC and Fortran or mnemonic (memory-aid) labels like ALGOL or assembly language. Remember to declare all labels at the beginning of the program.

The GOTO Statement

The GOTO statement will cause program execution to be immediately transferred to the statement that bears the GOTO label.

```
1ØØØ:Write ('Enter the date');
     Read (current_date);
     If current_date <= last_date
          Then goto 1ØØØ;
```

In general, we use numeric labels when dealing with the GOTO statement, and mnemonic labels when dealing with BREAK, CYCLE, and RETURN. In the preceding example the label 1000 is assigned to the WRITE statement. Then, based on a validation check in the IF statement, the program may *branch* back to repeat the data entry statements. You can see that this resembles BASIC's line numbers to some extent, but it should not become habitual for you "died in the wool" BASIC programmers. The same process can be handled much more elegantly by:

```
Repeat
     Write ('Enter the date');
     Read (current_date);
Until
     current_date > last_date;
```

The BREAK Statement

The BREAK statement is used to discontinue an iterative statement (FOR, WHILE, or REPEAT) and resume execution at the next statement following the end of the iterative structure. BREAK is useful for jumping out of an iterative structure after searching for a value in an index table. Here's a simple example:

```
Loop:For pointer := 1 to table_end
          Do   |
               table_entry := table [pointer];
               If search_value = table_entry
                    Then Break Loop;
                    Else count := count + 1;
               |.
```

Here again, a better approach should be used, taking advantage of one of Pascal's more specialized conditional programming statements.

```
pointer := 1;
While (pointer <= table_end)  and
      (search_value <> table [pointer])
         Do pointer := pointer + 1;
```

The RETURN Statement

The RETURN statement is used in much the same way as the BREAK statement. The difference is that RETURN is used to exit from a function or procedure instead of a loop.

The CYCLE Statement

The CYCLE statement is another in this category of escape clauses. Its purpose is to jump to the end of an iterative structure. This will cause the condition clause to be evaluated, and in the FOR statement, the control variable will be incremented or decremented, depending upon which direction the loop is going (TO or DOWNTO).

Summary

In this chapter we have added more powerful tools – in the form of statements used for decision-making and iteration – to our Pascal workbench. There are many elegant ways to use these tools; and many examples will appear in the following chapters.

Exercises

1. Solve the following Boolean expressions:

   A. $5 > 2$

   B. $(2 <= 5)$ and $(2 + 2 = 4)$

   C. $(0 > 1)$ or $(1 < 100)$

2. What is a compound statement?

3. What are the three loop structures for Pascal?

4. What are the differences between WHILE and REPEAT statements?

Solutions

1. A. True
   B. True
   C. True

2. A compound statement is one or more statements included within another statement. For example:

```
if a > Ø  then
   {
   b = b + a;
   c = c + 1;
   };
```

3. The three Loop structures for Pascal are: FOR...DO, WHILE...DO, REPEAT...UNTIL

4. The WHILE...DO loop tests the condition before executing the loop even once. The REPEAT...UNTIL always executes the statements in the loop first, then tests the condition afterwards. Also, the logic is reversed, for example:

```
WHILE  A = Ø    DO [...];
```

is similiar to

```
REPEAT  [...]  UNTIL A <> Ø;
```

except the first may not execute the statements at all.

4

Structured Data Types

Concepts

- Declaring an array
- Indexing and iteration
- Tables
- Sorting an array
- STRINGs and LSTRINGs as arrays
- Super array types
- Records and fields
- Static and dynamic variables
- The heap
- Pointers and linked lists
- Sets

Keywords

ARRAY, SUPER ARRAY, RECORD, WITH, NEW, DISPOSE, SET

Until now, we have been discussing simple data types and their related expressions and functions. These simple data types, discussed in chapter 2, can be any of the different types — INTEGER, CHAR or REAL and so on — but their important characteristic is that there is only one element of data per identifier name, regardless of its type.

In this chapter, we'll look at *structured* data types, which, in contrast to simple types, can be composed of many data elements, called components. We'll learn how to declare the structured types ARRAY, SUPER ARRAY, RECORD and SET. We'll also learn how to index the components in these structures in order to process them.

The ARRAY Structure

The *array* is a concept that is common to many programming

languages. It is used when there is a group of data elements that are all somehow related. By structuring the data elements in an array, we can take advantage of many powerful programming features to process data more efficiently than is possible with the simple data types. We'll show an example soon. In the meantime, let's see how arrays are declared and indexed.

Declaring an array

An ARRAY can be thought of as a chain in which each link represents one data element. Just as all the links in a chain are similar, so are all the components in an array. That's why when we declare an array in Pascal, the TYPE of its components must also be declared. The array is a variable, and is therefore declared in a VAR statement.

```
Var
    identifier :array[1..size] of (type clause);
```

Besides declaring the type of the components of the array, we must also declare how many components there will be in the array. This is done within brackets "["and"]" and is usually an integer subrange. In this case, the array size will be dictated by the value of "size," which would have to be defined in a previous CONST section. The low end of the subrange is usually "1" since normally there is no reason to declare an array starting with any component other than its first. Here are some examples of array declarations.

```
Var
    index_table    :array[1..24] of integer;
    sales_hist     :array[1..12] of real;
    access_table   :array[1..16] of boolean;
```

The variable "index_table" is declared as an array of 24 INTEGER types. Each one of these integers is called a *component* of the array. Likewise, the other arrays are declared in a similar way to have a certain number of components, of REAL and BOOLEAN types. There are as many different array types as there are types in general.

Indexing an Array

In order to make use of any of the components of an array, we must be able to isolate a particular component from the rest. This is accomplished through a process known as *indexing*. To reference a

component of an array, we use the array identifier, followed by a component number enclosed in brackets. In this way, a single component of the array structure can be be used as though it were a simple variable. Here is an example.

```
test_value := index_table [19];
```

Here, the value of the nineteenth consecutive component of the array with the name "index_table" is assigned to the variable "test_value" (presumably a compatible type). Figure 4-1 provides a graphic representation of this process.

Arrays and Iteration

In our chapter on program control, we mentioned that iteration is a valuable technique in dealing with structured variable types. Here's our old faithful FOR statement, just iterating away on sales history.

```
For index := 1 to 11
     Do sales_hist [index] := sales_hist [index+1];
```

This is typical of "rolling over" monthly sales history. The array named "sales_hist" contains total sales amounts for the past twelve months. The current month's sales are contained in the twelfth component of the array, and the oldest data is contained in the first component. As shown in Figure 4-2, each month's data is moved into the slot for the preceding month.

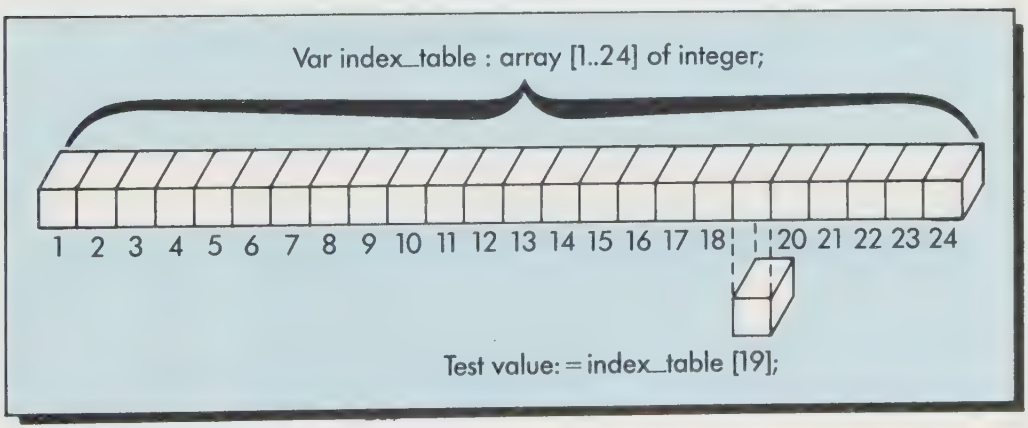


Figure 4-1. Indexing an array

Using Arrays for Tables

Sometimes a program application will involve the use of *soft switches* or *software toggles* to keep track of the system configuration. A large group of options can easily be represented as an ARRAY OF BOOLEAN type. Each component of the array can then represent some condition, and of course its value will be either TRUE or FALSE. Illustrated in Figure 4-3, this provides a quick method of screening access to an on-line system.

In our program, all we need to do is input the *menu selection* and then use the ARRAY OF BOOLEAN switches to provide controlled access to the system. Figure 4-4 shows another example.

You might want to go back to chapter 3, "Program Control," and take another look at the prime number sieve program shown in Figure 3-7. We slipped an ARRAY OF BOOLEAN types in on you in that example, knowing that we would be explaining arrays here.

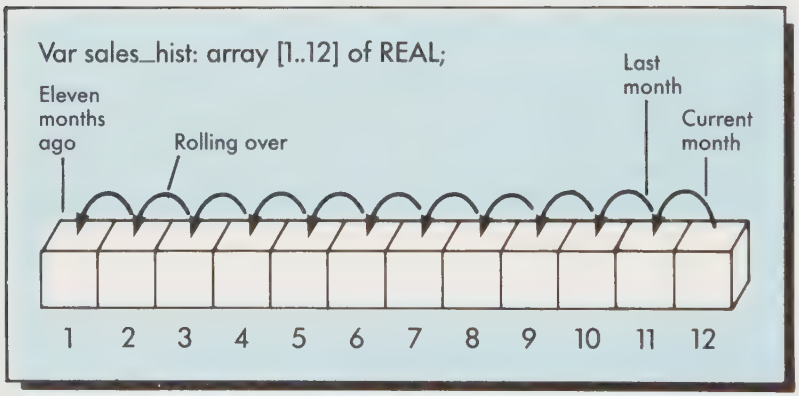


Figure 4-2. Iteration on an array

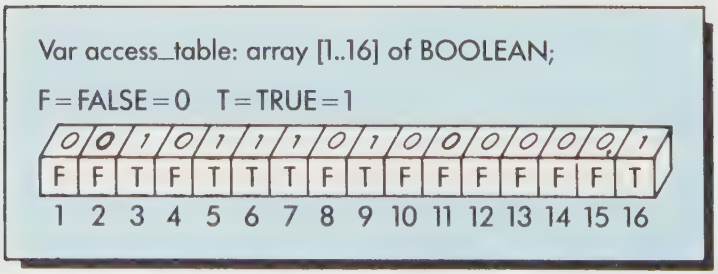


Figure 4-3. An array of BOOLEAN switches

Figure 4-4. Example of an array of BOOLEAN

```
{*****************************************************************
BOOLMENU.PAS    Menu access controlled through boolean table.
-----------------------------------------------------------------
}
Program Boolmenu (input,output);

Const
        size = 16;

Type
        range = 1..size;

Var
        menu_select    :range;
        access_table   :array[range] of boolean;

Value
        access_table[1]  := false;
        access_table[2]  := false;
        access_table[3]  := true;
        access_table[4]  := false;
        access_table[5]  := true;
        access_table[6]  := true;
        access_table[7]  := true;
        access_table[8]  := false;
        access_table[9]  := true;
        access_table[10] := false;
        access_table[11] := false;
        access_table[12] := false;
        access_table[13] := false;
        access_table[14] := false;
        access_table[15] := false;
        access_table[16] := true;

{
-----------------------------------------------------------------
        Main program.  Assuming a menu has been displayed, input
the selection.  Then use the boolean table to screen access.
-----------------------------------------------------------------
}
Begin
        Write ('Enter selection ----> ');
        Readln (menu_select);
```

```
        If access_table [menu_select]
                Then Writeln ('Access granted')
                Else Writeln ('Requires more authority');
End.
{
-------------------------------------------------------------------
Figure 4-4.     BOOLMENU.PAS    Example of array OF BOOLEAN
*******************************************************************}
```

Indexes of Enumerated Types

It is also possible to declare an array to be indexed by an enumerated type. This is one of the nicest features of Pascal. In other languages it would be necessary to set up some kind of complex coding scheme for data elements other than numbers and characters. But in Pascal, we can simply declare the valid components of an enumerated type. Figure 4-5 is an example of a program that uses an enumerated type array as a data table.

Figure 4-5. Example of an array as a table

```
{*****************************************************************
CHANGE.PAS      Change maker program.
-----------------------------------------------------------------
        This program demonstrates the use of arrays as tables.
In this case, change is made from a sales transaction.  The
program determines how many of each "coin" will make up the
change amount.
-----------------------------------------------------------------
}
Program Change (input, output);

Type
        coins = (dollar, quarter, dime, nickel, cent);

Var
        name                    { names of coins                }
                        :array [coins] of lstring(7);

        coinval                 { values of coins               }
                        :array [coins] of integer;

        sale,                   { sale amount                   }
        paid,                   { amount paid                   }
                        :real;

        change,                 { change in cents               }
        count,                  { count of currcoin needed      }
                        :integer;
```

```
        currcoin                { current coin                  }
                        :coins;

Value   { set names and values of coins                         }

        name[dollar]            := 'Dollar';
        name[quarter]           := 'Quarter';
        name[dime]              := 'Dime';
        name[nickel]            := 'Nickel';
        name[cent]              := 'Cent';

        coinval[dollar]         := 100;
        coinval[quarter]        := 25;
        coinval[dime]           := 10;
        coinval[nickel]         := 5;
        coinval[cent]           := 1;
{
-----------------------------------------------------------------
        Main program.  Input the amount of cash received and the
amount of the sale.
-----------------------------------------------------------------
}
Begin

        Write ('Enter sale amount ----> ');
        Readln (sale);

        While sale <> 0 Do  { terminate program when sale = 0 }
            {
            Write ('Enter amount paid ----> ');
            Readln (paid);
}
-----------------------------------------------------------------
                    Determine if change is due.
-----------------------------------------------------------------
{
            If sale = paid
                Then Writeln ('No change')

                Else If sale > paid
                        Then Writeln ('You still owe me',
                                      (sale - paid)   :6:2)
```

```
{
-------------------------------------------------------------------
                Compute and print the change.
-------------------------------------------------------------------
'
                    Else [
                        change := Round (1ØØ * (paid - sale));
                        Writeln ('Your change is',
                                    (change/1ØØ)    :6:2);
{
-------------------------------------------------------------------
       Determine how many of each coin are required and then
print a listing.
-------------------------------------------------------------------
!
'
                        For currcoin := dollar to cent Do
                            {
                            count := change Div coinval [currcoin];
                            change := change Mod coinval [currcoin];

                            If count <> Ø
                                Then [
                                    Write (count    :4, ' ',
                                            name [currcoin]);
                                    If count > 1
                                        Then Write ('s');
                                    Writeln;
                                    ];

                            ]; { end FOR...DO loop.          }
                        ];  { end IF...THEN
{
-------------------------------------------------------------------
                  Input the next sale amount.
-------------------------------------------------------------------
!
          Writeln;
          Write ('Enter sale amount ----> ');
          Readln (sale);
          ];                    { end WHILE...DO                }

End.                            { end PROGRAM                   }
{
-------------------------------------------------------------------
```

Figure 4-5. CHANGE.PAS Example of array as a table

The type "coins" is declared as an enumerated type consisting of the valid data elements "dollar," "quarter," "dime," "nickel," and "cent." Then, we declare two ARRAY type variables. First an array that will contain the names of the coins for display purposes, and then an array of their respective values. Notice the VALUE section where both the names and their values are defined.

In the program itself, we input a sales amount, and an amount paid. The program first determines whether there is any change due, or if the payment was insufficient. If there is change due, the program computes the amount and displays it. We've specified the way that the output is to look by including some editing parameters in the WRITELN statement.

```
Writeln ('Your change is',
                (change/100)    :6:2);
```

The "6:2" tells the program to print the change amount in the form

```
XXX.XX
```

The first number, "6," specifies the total number of columns to use for displaying the variable. The second number, "2," specifies the number of digits to the right of the decimal point. We'll see more examples of this kind of output formatting in future example programs.

Next, the program determines how many of each "coins" component should be given as change. The program takes advantage of Pascal's integer division operators, DIV and MOD, to simplify the calculations. The same values could have been obtained with the following statements.

```
count  := trunc (change / coinval [currcoin]);
change := change - count * coinval [currcoin];
```

The number of coins is printed whenever they are needed and an "s" is added when a plural is proper.

Figure 4-6 is an example of the output from the program Change.

Sorting an Array

Very often, a programming task involves sorting data elements into some useful sequence. This may be an organization based upon numeric size, or upon alphabetic order. With the enumerated type discussed earlier, we can sort data into just about any abstract logical sequence: by color, for example. That's because each enumerated value equates to a numeric value determined by its order in the declaration. The first enumeration is equated with numeric zero. The second with numeric one, and so on.

Figure 4-7 is an example of a simple Pascal program that inputs some INTEGER type variables into an array and then sorts the array into ascending numeric sequence and prints it out.

Figure 4-6. Output from the Change program

```
Enter sale amount ----> 4.49
Enter amount paid ----> 5.ØØ
Your change is  Ø.51
   2 Quarters
   1 Cent

Enter sale amount ----> 12.2Ø
Enter amount paid ----> 1Ø.ØØ
You still owe me  2.2Ø

Enter sale amount ----> 14.Ø4
Enter amount paid ----> 2Ø.ØØ
Your change is  5.96
   5 Dollars
   3 Quarters
   2 Dimes
   1 Cent

Enter sale amount ----> 1.ØØ
Enter amount paid ----> 1.ØØ
No change

Enter sale amount ----> Ø

A>
```

Figure 4-7. Example of sorting an array

```
{*****************************************************************
ARRYSORT.PAS    Example of sorting an array.
-----------------------------------------------------------------
        This program takes any 1Ø integers, entered in any order
and sorts them into ascending sequence using an "exchange sort"
technique.
-----------------------------------------------------------------
}
Program Arrysort (input,output);

Var
        numbers         :array[1..1Ø] of integer;

        index,
        temp,
                        :integer;

        exchanged       :boolean;

Const
        size = 1Ø;
{
-----------------------------------------------------------------
            Main program.  Enter 1Ø integer values.
-----------------------------------------------------------------
}
Begin
        Writeln ('Enter 1Ø integers in any order');
        For index := 1 to size
                Do Readln (numbers [index]);

{
-----------------------------------------------------------------
        Scan through the array.  Exchange the position of adjacent
components that are out of sequence, using a temporary variable.
Remain in the REPEAT structure until no exchanges are made.
-----------------------------------------------------------------
}
        Repeat
            exchanged := false;

            For index := 1 to size-1 Do
                If numbers [index] > numbers [index+1]
                    Then [
                        temp := numbers [index];
                        numbers [index] := numbers [index+1];
                        numbers [index+1] := temp;
                        exchanged := true;
                        ];
```

```
        Until not exchanged;

{-------------------------------------------------------------------}
                    Now, print out the table
{-------------------------------------------------------------------}

        Writeln ('Ascending sequence');

        For index := 1 to size Do
                Write (numbers [index]);

End
{-------------------------------------------------------------------}
```

Figure 4-7. ARRYSORT.PAS Example of sorting an array

This program provides a good example of what indexing is all about. We have declared an INTEGER array called "numbers" which will be used to contain the values to be sorted. The INTEGER type "index" will be used to accesss individual components in the array.

There are basically three parts to the Arrysort program.

Enter the data.

Sort the data.

Print the data.

Each of these parts employs the use of an iterative structure to index the array components. The FOR statement is usually the principal mechanism for implementing this structure with arrays, since it can be incremented or decremented within some range of values. One thing that FOR doesn't do is *step* a specified increment each time through the loop, like BASIC's FOR...NEXT statement. The increment is always 1 because it is actually performed by the two functions SUCC (for incrementing) and PRED (for decrementing). If it is really necessary to *step through* a FOR statement at some other interval, it must be done using a *secondary variable* derived from the FOR statement's control variable. This kind of stepping should not be necessary in Pascal, if the programmer has really thought out the structure of the data that will be processed.

The sorting algorithm is really quite simple. It's called a *bubble sort*, sometimes an *exchange sort*, because of the way it works: the highest value "bubbles" to the top of the array.

Here's how it works. Two adjacent components in the array are compared. If the first one is numerically greater than the second, they are exchanged in the array. Then the next two are tested, and so on. In

order to exchange two array components we must use an intermediate variable, which we call "temp" in this program, to temporarily save one of the components. Since an array component can only be shifted one position at a time, several passes through the exchange loop may be necessary depending upon the original sequence of the components. The worst case would be if the smallest value was at the top and the largest value at the bottom of the array.

Each time the program is about to enter the exchange loop, it sets the BOOLEAN type variable "exchanged" to FALSE, assuming that no exchanges will be done on this pass through the loop. In the true statement of the IF, this flag is set to TRUE, indicating that an exchange has occurred. The exchange loop is then terminated by the condition clause of the REPEAT statement, since we are looking for a TRUE value here to stop the REPEAT.

Perhaps this can be visualized more clearly by using reverse logic. The loop should end if there have been no exchanges, that is, when "exchanged" = FALSE. So... NOT (FALSE) = TRUE, and the loop is terminated. Figure 4-8 is a flowchart of this sort algorithm.

Finally, the sorted array is printed, again using a loop to index the components. Each component is printed separately. Using the WRITE statement instead of WRITELN causes them all to be printed on the same line.

This ends our general introduction to arrays. Now we will take a look at some of the specific types of arrays used in Pascal programs.

STRING and LSTRING

We have bent the rules slightly in this book, in order to provide a more meaningful approach to learning Pascal for the IBM, in that we have included the STRING and LSTRING types with the simple data

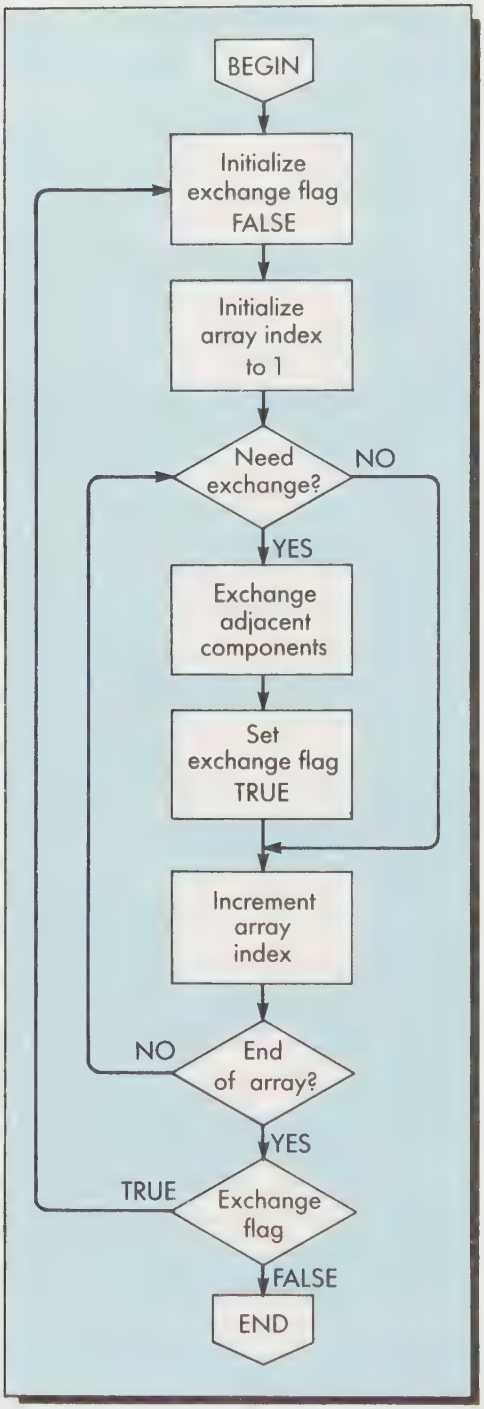


Figure 4-8. Flowchart ot the sort algorithm

types described in Chapter 2. One reason for this is that they are similar to BASIC language strings. But in reality, STRINGs and LSTRINGs are perfect examples of structured data types.

The technical declaration for STRING in Pascal is

```
Packed array[1..N] of char
```

But since STRINGs come up so often in programming, IBM decided to shorten the declaration to simply:

```
String(N)
```

This is accomplished through a predeclared *super type*, which we'll discuss further in this chapter. The LSTRING type is also predeclared in this manner.

Length of STRING and LSTRING

In IBM Pascal, the length of the STRING is represented here by "N." During compilation, the length will be determined by an INTEGER value. This is why the maximum length of a STRING should be 32,767. (IBM says it's 32,766. Perhaps they wanted it to be an even number.) Actually, this is the maximum length of any structured type in IBM Pascal. When you think about it, that's quite a long string. It's over 6,500 words, just about 20 pages of unbroken text.

The LSTRING type is a special structure similar to a STRING that also contains its own current length. Remember, the length is contained in the first byte of the LSTRING, so the maximum size for an LSTRING type is 255 characters. This is more than enough for most simple string applications. Figure 4-9 shows the internal representation of STRING and LSTRING types.

There are some other advantages to using LSTRINGs. IBM has several predeclared procedures for manipulating them. Also, an LSTRING type may be used directly in a condition clause; a useful feature, as you will notice in our example programs.

One situation in which we like to use an LSTRING involves testing data as it is input from the user. It's nice to have the user type "end" when there is to be no further input. This is especially convenient if we are inputting alphabetic data, like someone's name. Then, we can use the following approach.

```
Var
     name      :lstring(1Ø);

Write ('Enter name ----> ');
Readln (name);

While name <> 'end'
     Do .....
```

Here, the processing would be controlled by the WHILE...DO structure, which is repeated as long as the name that was input is not "end." We couldn't do this so easily using a STRING, because the entire length of a STRING is involved in all operations, while only the current length of an LSTRING is used.

If you're still having trouble with STRINGs and LSTRINGs, then perhaps this simple program example will shed some light on them for

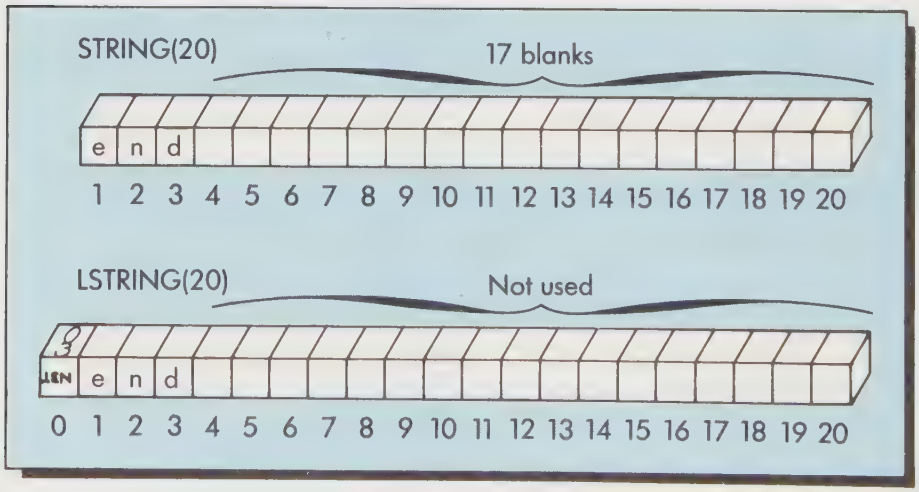


Figure 4-9. STRING and LSTRING representation

you. Figure 4-10 is a STRING and LSTRING demonstration program. It is an endless loop program that will not end until you type Ctrl C.

Figure 4-10. Example of STRING and LSTRING

```
{*************************************************************
STRDEMO.PAS     STRING and LSTRING Demo Program.
-------------------------------------------------------------
        This program demonstrates the difference between STRING
and LSTRING types.  The STRING is always the same length, while
the length of the LSTRING can vary.
-------------------------------------------------------------}

Program Strdemo (input,output);

Const
        max = 8Ø;

Var
        str_type        :string(1Ø);
        lstr_type       :lstring(1Ø);
        count           :integer;

{-------------------------------------------------------------
        Main program.  Input both STRING and LSTRING values,
output each 8Ø times.
-------------------------------------------------------------}

Begin
    While true DO [
        Write ('Enter STRING(1Ø) value ----> ');
        Readln (str_type);
        Writeln;
        For count := 1 to max
                Do Write (str_type);

        Writeln;
        Write ('Enter LSTRING(1Ø) value ---> ');
        Readln (lstr_type);
        Writeln;
        For count := 1 to max
                Do Write (lstr_type);
End.    ];

{-------------------------------------------------------------
Figure 4-1Ø     STRDEMO.PAS     Example of STRING and LSTRING
*************************************************************}
```

Notice what happens when you type fewer than 10 characters. The STRING version still outputs 10 characters, even if some of them are blank. The LSTRING version on the other hand, outputs only the current length of the LSTRING. Figure 4-11 shows the output from the program shown in Figure 4-10.

Figure 4-11. Difference between STRING and LSTRING

```
A>strdemo
Enter STRING(10) value ----> 123

123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123
123       123       123       123       123       123       123       123

Enter LSTRING(10) value ---> 123

12312312312312312312312312312312312312312312312312312312312312312312312312312312
31231231231231231231231231231231231231231231231231231231231231231231231231231231
23123123123123123123123123123123123123123123123123123123123123123123123123123123
```

Sorting an Array of LSTRING Type

Let's suppose an array contains people's names, and we want to sort them into alphabetic order. This is going to be quite different from sorting numbers...right? Wrong! The program is almost identical, only the names have been changed. Figure 4-12 is an example of how sorting an array of LSTRINGs is just like sorting numbers.

Figure 4-12. Example of sorting LSTRINGs

```
{*****************************************************************}
NAMESORT.PAS    Sorting an ARRAY OF LSTRING.
{---------------------------------------------------------------}
        This program demonstrates how LSTRINGs can be sorted
just like numeric values.
{---------------------------------------------------------------}

Program Namesort (input,output);

Const
        size = 1Ø;

Type
        first_name = lstring(15);

Var
        names     :array[1..size] of first_name;

        index     :integer;

        temp      :lstring(15);

        exchanged :boolean;

{---------------------------------------------------------------}
        Main program.  Enter a list of names and the program
will sort them using the same exchange sort method.
{---------------------------------------------------------------}

Begin
        Writeln ('Enter ', size,' names in any order');
        For index := 1 to size
                Do Readln (names [index]);

{---------------------------------------------------------------}
            Exactly the same idea as sorting numbers.
{---------------------------------------------------------------}

    Repeat
        exchanged := false;

        For index := 1 to size-1 Do
                If names [index] > names [index+1]
                    Then [
                        temp := names [index];
                        names [index] := names [index+1];
                        names [index+1] := temp;
                        exchanged := true;
                        ];
```

```
    Until not exchanged;
'
{-------------------------------------------------------------------}
{             Now, print out the list in sequence                   }
{-------------------------------------------------------------------}
;
        Writeln ('Ascending sequence');
        For index := 1 to size Do
                Writeln (names [index]);
End.
{
{-------------------------------------------------------------------}
Figure 4-12.    NAMESORT.PAS    Example of sorting LSTRINGs
*******************************************************************}
```

We simply changed the array variable "numbers" to one called "names," and declared "names" as an array of LSTRING type. This allows us to easily compare two LSTRINGs of different lengths. The names are input just as the numbers were. The sorting uses the same algorithm to exchange two adjacent name components. The output section is pretty much the same also: the only change we made was to use

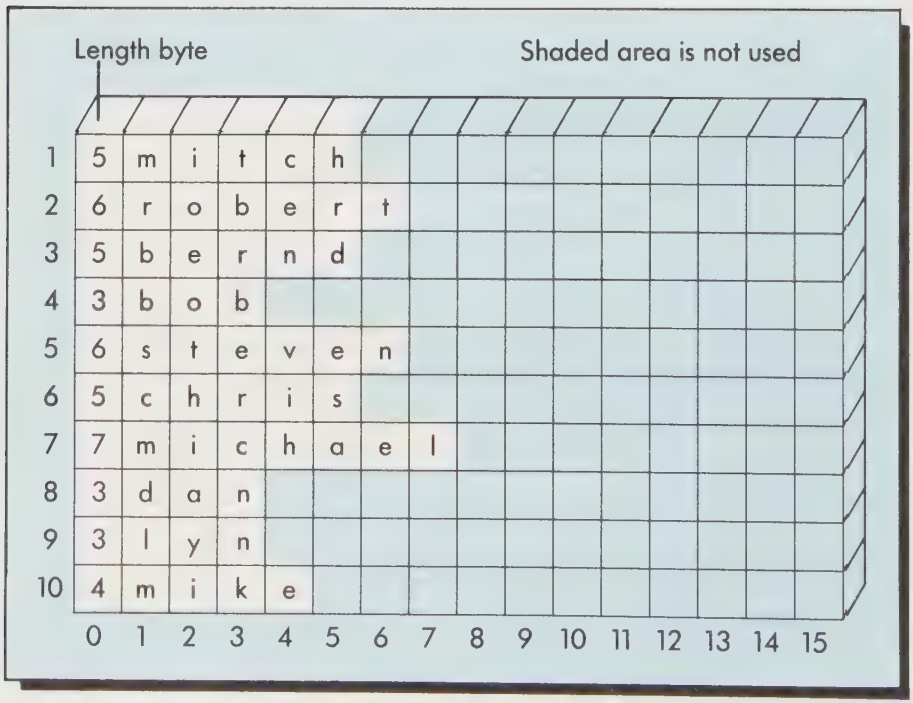


Figure 4-13. An array of LSTRING type

WRITELN instead of WRITE to make each name appear on a line by itself.

Notice the power of this structure, which is shown in Figure 4-13. We can refer to each of the components in the array, manipulate it, and use it to generate output, all within a few statements. Again the predominant structure is the FOR statement, used to index the array of names. The control variable "index" is used to point to one component of the array so that it may be processed.

SUPER ARRAY Type

In the previous examples, we have shown the standard form of arrays in Pascal. IBM has added a special type declaration to their version of Pascal called the *super array* type. The key to understanding this type is to realize that it is not a variable. Think of the a super array as a *type declaration* that allows arrays to be declared in simpler, perhaps more familiar, form. An example that will tie this together involves our new acquaintances STRING and LSTRING. The complete declaration for a STRING variable in standard Pascal looks like this.

```
Var
    name      :packed array[1..15] of char;
```

That looks reasonable you say, but what's the "packed" do? It tells the compiler that the characters in the array "name" will be stored one character per byte. Otherwise, the compiler would use a whole 16-bit word for each character. That's because the compiler allocates space for variables in increments of two bytes at a time. (That's why the memory addresses of the variables in a program will always be even numbers.)

The numbers inside the brackets serve to define the maximum possible number of characters that can be stored in the array, in this case 15. And of course the "of char" tells the compiler that these will be ASCII characters or their equivalents.

However, this packed array format is rather unwieldy. The IBM people realized that if your program had a lot of string variables in it, you would have to do a lot of typing just to get them declared. So they invented the SUPER ARRAY.

Declaring a SUPER ARRAY

The SUPER ARRAY type is simply an array type declaration with the

upper boundary set up as a "plug-in" parameter. Normally we would have to declare a group of STRINGs like this:

```
Var
     name      :packed array[1..25] of char;
     address   :packed array[1..25] of char;
     city      :packed array[1..2Ø] of char;
     state     :packed array[1..2] of char;
     zipcode   :packed array[1..9] of char;
     areacode  :packed array[1..3] of char;
     telephone :packed array[1..7] of char;
```

However, using the SUPER ARRAY type declaration, it can be done like this:

```
Type
     string    :super packed array[1..*] of char;

Var
     name      :string(25);
     address   :string(25);
     city      :string(2Ø);
     state     :string(2);
     zipcode   :string(9);
     areacode  :string(3);
     telephone :string(7);
```

The compiler knows that the type "string" will need a parameter in parentheses to fill in for the "*" in the TYPE declaration. In IBM Pascal, STRING is a predeclared super type, and the TYPE statement is not required.

We can also set up our own SUPER ARRAYs to handle special situations. If the application at hand only uses one of a particular array type, then it makes no sense to go through the extra steps of declaring it a SUPER ARRAY. Only when there are to be several arrays of the same type, but different sizes, does the SUPER ARRAY concept help.

The Sales Update Program

Suppose we are writing a sales update program. Its job will be to update sales totals that are represented in memory as arrays. Why arrays? Because we want to break out the sales information into 10 merchandise categories. We also want to save the sales statistics for the current month, current fiscal quarter, and year to date. Figure 4-14 is an example of how to use SUPER ARRAYs for this application.

Figure 4-14. Example of array of REAL

```
{*****************************************************************
SALESUPD.PAS    Sales Update Program.
-----------------------------------------------------------------
        This program demonstrates the use of arrays to store
numeric data.  There are three arrays used in this program, one
each for "current month," "current quarter," and "year to date."
-----------------------------------------------------------------}

Program Salesupd (input,output);

Const
        cats = 1Ø;              { number of categories }

Type
        sales_history = super array[1..*] of real;

Var
        current_month,
        current_quart,
        year_to_date
                        :sales_history (cats);

        category
                        :Ø..cats;

        amount
                        :real;
{
-----------------------------------------------------------------
Main program.  Enter category code and amount.  As long as the
category code is not zero, update the proper component of the
sales history arrays.
-----------------------------------------------------------------}

Begin

        Writeln ('Enter category and amount');
        Readln (category, amount);

        While category <> Ø Do
                {

                current_month [category] :=
                    current_month [category] + amount;

                current_quart [category] :=
                    current_quart [category] + amount;
```

```
                year_to_date [category] :=
                    year_to_date [category] + amount;

                Readln (category, amount);
                ];
;
-------------------------------------------------------------
          Now, print out the sales history arrays.
-------------------------------------------------------------

'
       For category := 1 to 10 cats Do
              Writeln ('Category ', category,
                     current_month [category],
                     current_quart [category],
                     year_to_date  [category]);
End.
!
'
-------------------------------------------------------------
Figure 4-14.    SALESUPD.PAS    Example of array of REAL
*************************************************************}
```

There are some things to look at here. The type declaration for the type "sales_history" allows the three arrays to be declared in the VAR statement much the same as STRING and LSTRING. Notice the variable "category" is a subrange from 0 through 10 instead of 1 through 10. This allows us to enter category zero to indicate the termination of the data entry phase.

The RECORD Type

We have just learned about one of the most powerful data structures available in most programming languages today. The array provides a means of organizing data for ease of input, processing, and output. But, there is one thing that limits the array's use: all of the components of the array must be of the same type. For instance, in the preceding sales update program, (Figure 4-14), every array component represents sales volumes in dollars and cents.

Let's enhance our sales update program. Let's suppose we wanted to do more than just update the sales amount. We might also want to keep a count of how many times sales are updated in each category. Also, we might want to know what the amount of the last sale was, and keep a running average sale amount for each category. And just to make things really neat, let's include a description of each category, as an LSTRING, which can be used on printouts.

The RECORD Layout

We could set up separate arrays for each of these requirements we've decided upon, but this would be awkward. Pascal provides a better alternative: the RECORD type structure. Figure 4-15 illustrates how we'll set up a RECORD structure.

We are dealing with three different data types, all related to the same category. We have the category "name" which will consist of several CHAR type elements, the "count" which will be an INTEGER, and the sales totals themselves which are REAL types. Figure 4-16 shows how the RECORD type is used in a program.

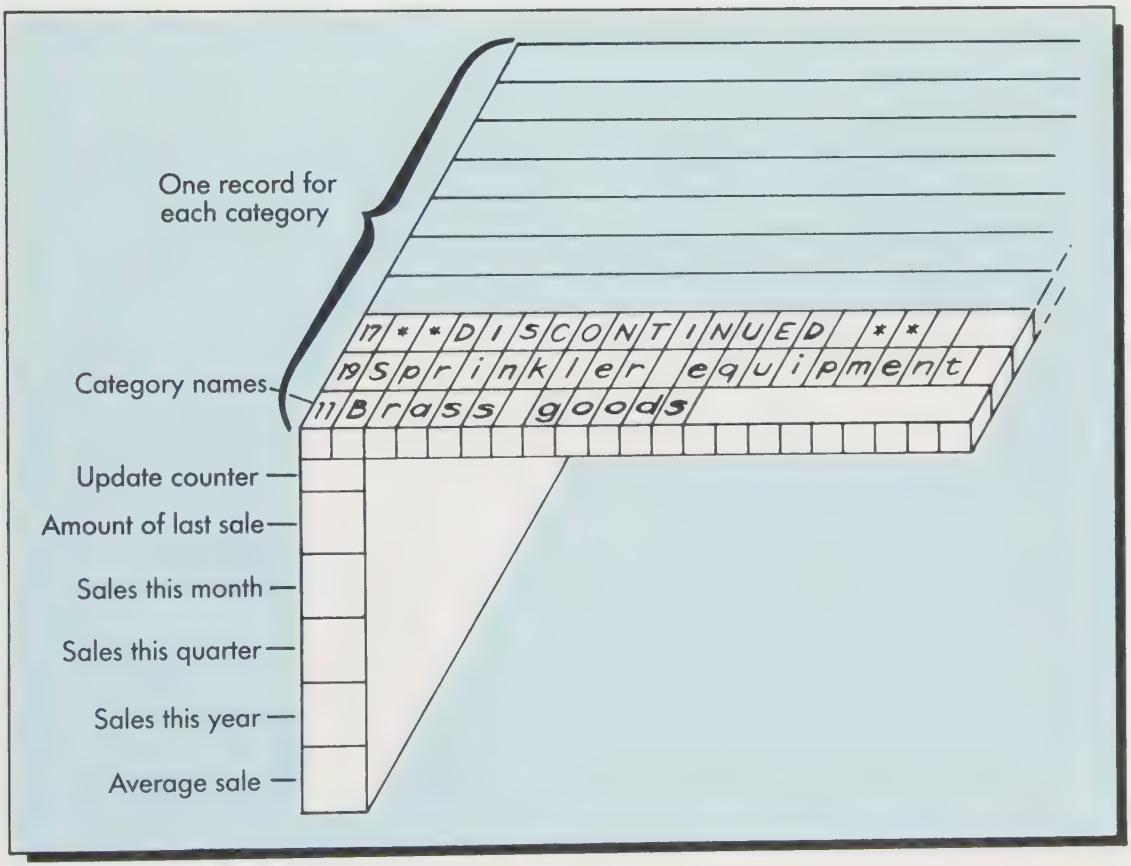


Figure 4-15. Sales statistics record layout

Figure 4-16. Sales update using RECORD type

```
{*****************************************************************}
SALESREC.PAS    Sales update using a record.
-----------------------------------------------------------------
        This program illustrates the use of RECORD type structure
to facilitate the sales update operation.
-----------------------------------------------------------------
}
Program Salesrec (input,output);

Const
        maxcat = 1Ø;                { maximum number of categories }
        width = 8Ø;                 { width of screen }

Type                                { <---- sales stats record
                                      declaration }
        stats =
            record
                name
                                    :lstring(2Ø);
                count
                                    :integer;
                last,
                month,
                quart,
                year,
                aver
                                    :real;
            end;

Var
        sales                       { <---- an array of records     }
                            :array[1..maxcat] of stats;

        cat
                            :Ø..maxcat;

        col
                            :1..width;

        amount
                            :real;
```

```
{
-------------------------------------------------------------------
     This value section contains the product descriptions
-------------------------------------------------------------------
}
Value

        sales [1] .name := 'Brass goods         '.
        sales [2] .name  = 'Sprinkler equipment '.
        sales [3] .name := '** DISCONTINUED **  '
        sales [4] .name := 'Rubber goods        '.
        sales [5] .name := 'Insulation          '.
        sales [6] .name := 'Used Equipment      ';
        sales [7] .name := 'Toilet Seats        ';
        sales [8] .name := '** DISCONTINUED **  ';
        sales [9] .name := 'Air conditioning    '.
        sales [10].name := 'Copper Products     '.
{
-------------------------------------------------------------------
  Main program.  Input sales transactions. category and amount
-------------------------------------------------------------------
}
Begin

        Writeln ('Enter category and amount on one line');
        Writeln ('  Enter zeros to end').
        Writeln;

        Readln (cat, amount);
{
-------------------------------------------------------------------
        Update the category specified with the amount.  Each
field is updated separately using the ".FIELDNAME" notation.
-------------------------------------------------------------------
}
        While cat <> 0
            Do   [

                sales [cat] .count :=
                    sales [cat] .count + 1;

                sales [cat] .last :=
                    amount;

                sales [cat] .month :=
                    sales [cat] .month + amount;

                sales [cat] .quart :=
                    sales [cat] .quart + amount;
```

```
                sales [cat] .year  :=
                    sales [cat] .year  + amount;

                sales [cat] .aver :=
                    sales [cat] .month /
                        Float (sales [cat] .count);

{
-------------------------------------------------------------
                And, input the next transaction
-------------------------------------------------------------
}
                Readln (cat, amount);
                };
{
-------------------------------------------------------------
            Now, list the totals for each category.
-------------------------------------------------------------
}
        Writeln;
        Write   ('DESCRIPTION        COUNT      LAST');
        Write   ('     MONTH     QUART      YEAR');
        Writeln ('   AVERAGE');

        For col := 1 to width Do  { <---- horizontal line.      }
                Write (chr (196));

        Writeln;

        For cat := 1 to maxcat Do
                Writeln (sales [cat] .name :2Ø,
                         sales [cat] .count :4,
                         sales [cat] .last  :1Ø:2,
                         sales [cat] .month :1Ø:2,
                         sales [cat] .quart :1Ø:2,
                         sales [cat] .year  :1Ø:2,
                         sales [cat] .aver  :1Ø:2        );

End.
{
-------------------------------------------------------------
Figure 4-16.    SALESREC.PAS    Sales update using RECORD type
*************************************************************}
```

Sales Update Using a RECORD

The first thing to look at in Figure 4-16 is the TYPE statement. This is another feature that makes Pascal a unique language. You can define just about any kind of structure imaginable, (and some that you can't imagine). Here, we have declared a data type called "stats" as a RECORD

consisting of several *fields*. The fields are declared following the word RECORD. Each one is declared with its respective TYPE. First, the "name" is declared as an LSTRING of length 20. Then the "count" is declared as an INTEGER. Finally, the "dollars and cents" amounts are all declared as REAL types. The RECORD declaration is terminated with an END statement.

Now that the "stats" type has been declared, we can declare a variable of that type. Since we know there are going to be several categories and we will want the same information processed for each one, what better structure than an ARRAY OF RECORDS? This is a very common data structure in Pascal. It allows indexed processing of multiple records. It also provides for iterative processing. So we'll declare a variable "sales" to be an array of the "stat" type records.

Accessing Fields within a Record

A particular field that has been declared in the RECORD type "stats" is accessible by simply appending a period and the field name to the record name. Of course, the name "sales" refers to an ARRAY OF RECORDS, each record representing a sales category. This means that the particular record must be indexed before the field can be isolated. This is done just as in any other array, by using the bracket notation "[" and "]". So for example, the "count" field in the fifth record would be identified by:

```
sales [5] .count
```

We'll use the "category" entered as input data to index the proper component of the array. This will also be used to terminate the data entry if its value is zero. To guarantee a valid index, we will declare the category as a subrange type. This will produce a runtime error if anything outside the range 0..maxcat is entered for the category.

For this example, the category names are declared in a VALUE statement. These could also be entered from the keyboard if so desired. But, assuming that the category names won't change that often, we'll *hard-code* them into the program. This means that if you do want to change the category names, you will have to modify and re-compile the program.

The program operation is really quite similar to our original sales update example. The category and amount are entered for the first time outside the WHILE structure. The program will then enter the WHILE structure and loop there as long as the category is not equal to zero. In

the structure, the variable "cat" entered as the category is used to index the array of sales statistics records. Then, each of the proper fields is updated. The "count" is incremented (we could also use the SUCC function for this). The current amount entered is stored in the "last_sale" field. The "month," "quarter," and "year" fields are all updated. And of course, the "average" sale is re-computed.

When a zero category is entered, the program falls out of the WHILE structure and the array is printed. First a "heading line" is printed (in pieces so it fits in this book). Next, a "heading underline" is printed in a FOR statement using the alternate character set. Finally, each of the fields is printed in the order they appear in the WRITELN statement. Notice the formatting parameters that will control the field size on the printout. With a little imagination, your output can take on the look of a professional product as shown in Figure 4-17.

Figure 4-17. Output from the sales update program

```
*******************************************************************************
DESCRIPTION         COUNT      LAST      MONTH      QUART       YEAR    AVERAGE
-------------------------------------------------------------------------------

Brass goods             3      2.5Ø     17.49      17.49      17.49       5.83
Sprinkler equipment     1    25Ø.ØØ    25Ø.ØØ     25Ø.ØØ     25Ø.ØØ     25Ø.ØØ
** DISCONTINUED **     Ø      Ø.ØØ      Ø.ØØ       Ø.ØØ       Ø.ØØ       Ø.ØØ
Rubber goods            1     16.5Ø     16.5Ø      16.5Ø      16.5Ø      16.5Ø
Insulation              3      1.21      2.59       2.59       2.59       Ø.86
Used Equipment          Ø      Ø.ØØ      Ø.ØØ       Ø.ØØ       Ø.ØØ       Ø.ØØ
Toilet Seats            Ø      Ø.ØØ      Ø.ØØ       Ø.ØØ       Ø.ØØ       Ø.ØØ
** DISCONTINUED **     1     -9.49     -9.49      -9.49      -9.49      -9.49
Air conditioning        Ø      Ø.ØØ      Ø.ØØ       Ø.ØØ       Ø.ØØ       Ø.ØØ
Copper Products         Ø      Ø.ØØ      Ø.ØØ       Ø.ØØ       Ø.ØØ       Ø.ØØ

-------------------------------------------------------------------------------
Figure 4-17.    SALESOUT.TXT    Output from sales update program
*******************************************************************************
```

The WITH Statement

Now we are getting into the heart of Pascal's ability to manage all sorts of different data structures. The ARRAY OF RECORDS demonstrated in the last example will suit many simple programming situations. As you can see, it is easy to use, once you have thought out the requirements of the particular structure. In this section, we will see how things can be made even easier using the WITH statement. The general form is:

```
With (record_name) Do ..
```

Look back at the previous example (Figure 4-16). Notice how in every instance where the ARRAY OF RECORDS is referenced, the first thing that appears is the indexed component of the array "sales [cat]." Then comes the field identifier ".name," etc. Well, it can get downright boring having to type this same thing on every line. It also increases the chance of a typing error. IBM Pascal has a solution for this problem. Using the WITH statement, we need only index the component once. Then, as long as the program remains within the structure of the WITH statement, the fields may be referenced by their field name only.

Sales Update Using the WITH Statement

Figure 4-18 is a new version of the sales update program using the WITH statement. Compare it to **Figure 4-16** and you'll see how valuable the WITH structure can be.

Figure 4-18. Sales update using WITH structure

```
{*************************************************************
SALESWIT.PAS
        Sales Update Using WITH Structure.
-------------------------------------------------------------
        This program inputs sales transaction categories and
amounts and update the sales statistics.  The WITH structure
is used to simplify the array and record combination.
-------------------------------------------------------------
}
Program Saleswit (input,output);

Const
        maxcat = 10;
        width = 80;

Type
        stats =
            record
                name
                                :lstring(20);
                count
                                :integer;
                last,
                month,
                quart,
                year,
                aver
                                :real;
            end;
```

```
Var
        sales
                        :array[1..maxcat] of stats;

        cat
                        :Ø..maxcat;

        col
                        :1..width;

        amount
                        :real;

Value
        sales [1] .name := 'Brass goods         ';
        sales [2] .name := 'Sprinkler equipment ';
        sales [3] .name := '** DISCONTINUED **  ';
        sales [4] .name := 'Rubber goods        ';
        sales [5] .name := 'Insulation          ';
        sales [6] .name := 'Used Equipment      ';
        sales [7] .name := 'Toilet Seats        ';
        sales [8] .name := '** DISCONTINUED **  ';
        sales [9] .name := 'Air conditioning    ';
        sales [1Ø].name := 'Copper Products     ';
{
-------------------------------------------------------------------
            Main program.  Input category and amount.
-------------------------------------------------------------------
}
Begin

        Writeln ('Enter category and amount on one line');
        Writeln ('  Enter zeros to end');
        Writeln;

        Readln (cat, amount);
{
-------------------------------------------------------------------
        Update the sales category using the WITH structure.  In
this way, only the field names are required.
-------------------------------------------------------------------
}
        While cat <> Ø Do
            [
            With sales [cat] Do { <---- WITH statement
                [
                count := count + 1;
                last := amount;
                month := month + amount;
                quart := quart + amount;
```

```
                year  := year  + amount;
                aver := month / Float (count);
                );

            Readln (cat, amount);
            );
{
-------------------------------------------------------------------
                    Now display the totals
-------------------------------------------------------------------
}
        Writeln;
        Write   ('DESCRIPTION        COUNT      LAST');
        Write   ('     MONTH     QUART      YEAR');
        Writeln ('   AVERAGE');

        For col := 1 to width Do
            Write (chr (196));
        Writeln;
{
-------------------------------------------------------------------
        Here again the WITH structure is used.  Also notice the
formatting specifications for each of the fields.
-------------------------------------------------------------------
}
        For cat := 1 to maxcat Do
            With sales [cat] Do
                Writeln (name :2Ø,
                         count :4,
                         last  :1Ø:2,
                         month :1Ø:2,
                         quart :1Ø:2,
                         year  :1Ø:2,
                         aver  :1Ø:2 );
End.
{
-------------------------------------------------------------------
Figure 4-18.    SALESWIT.PAS    Example of the WITH structure
*******************************************************************}
```

You will notice that this program looks a lot more concise. In fact, it is smaller, both at the source code and executable levels. It is also easier to see what's happening by looking at the listing.

There are two places in the program where we have used the WITH statement. The most obvious is in the WHILE structure where the actual updating process is done. We have inserted the statement:

```
With sales [cat] Do
```

and then removed all other references to the array or the index for the remainder of the WITH statement. Instead, we simply use the field names in the processing. Now isn't that nice?

The other place we used the WITH statement is in the section of the program that prints out the sales statistics array. Here, we have inserted the WITH statement in the DO clause of the FOR structure that prints out the data. This will take care of the array indexing, giving us one record at a time. In the balance of the statement, each of the fields in the record is output by its name only. The formatting still applies as before.

Dynamic Allocation of Variables

All of the data types that we have explored so far have been of a fixed size. The size has been determined either explicitly in the declaration, or through the use of some structured constant. When we declare an array of 10 REAL types like this:

```
Var
    numbers
            :array[1..10] of real;
```

there is no doubt that there will be exactly 10 components to the array identified as "numbers." The compiler sets aside the space in memory for the array, and keeps track of that memory address in the symbol table. Whenever the array is referenced by a statement within the program, the compiler "knows" where the array is going to be, and can compile actual memory addresses into the program. Variables like these are called *statically allocated*. They can be considered as much a part of the program as the instructions.

It is, however, sometimes hard to know, at the time you write the program, just how many components will be necessary in an array. There are program applications where the number of components can vary greatly between different executions. For example, think of entering daily sales transactions. Some days might be slow, and there would be only a few transactions. On busy days, there would be more transactions.

If static allocation was the only feature available, then you would have to be prepared to handle the "worst case" – largest number – of transactions predicted. The various arrays would need to be declared as large as possible without risking running out of memory. Then, most of the time, you'd find you hadn't used anywhere near the whole array. Of course the real hitch is that someday the memory won't be large enough. If you need two or more such arrays then the problem can become very complicated.

Variables on the Heap

Pascal's answer to this problem is *dynamic allocation* of variables from an area of memory called the *heap*. The heap is just that... all the memory that is not currently being used for either program instructions or static data. At runtime, portions of the heap are allocated to those variables that require it through the use of the procedure NEW, and de-allocated through the use of DISPOSE. A special data type called a *pointer* is used to access variables on the heap.

The Pointer Type

Since the location of these dynamically allocated variables is not known at compile time, the standard techniques of indexing and iteration can not be applied. The notation using brackets to select an array component cannot be used in the usual way either. That's where the pointer comes into the picture. Internally, the pointer is just another 16-bit quantity. Treated as a WORD, it represents the "offset" part of the address of the memory location that contains the data. This pointer may "point" to any of the 64K memory locations in the data segment. (To learn more about the IBM PC's memory and how to address it, see *Assembly Language Primer for the IBM PC and XT*, by Robert Lafore, (New York: Plume/Waite, New American Library, 1984).

The small program shown in Figure 4-19 will give you the basic picture of how pointers work. The program will input names of no more than 10 characters in length, and allocate part of the heap to store them, until the word "end" is entered.

Figure 4-19. Example of POINTERs and the HEAP

```
{*****************************************************************
DYNA.PAS        Dynamic name entry program.
-----------------------------------------------------------------
        This program illustrates the use of POINTER types and
dynamic allocation of variables on the HEAP.
-----------------------------------------------------------------}

Program Dyna (input,output);

Type
        name = lstring(1Ø);      { <---- first declare the type  }

Var
        curr_ptr                 { <---- then declare pointer    }
                        :^name;
        entry
                        :name;
```

```
{-------------------------------------------------------------------
        Main program.  Names of up to 10 characters can be
entered until your system runs out of space on the HEAP.
-------------------------------------------------------------------}

Begin

        Writeln ('Enter names forever');
        Readln (entry);

{-------------------------------------------------------------------
        Use the NEW procedure to define a new pointer value.
Display the pointer value and then assign the variable.
-------------------------------------------------------------------}
{
        While entry <> 'end' Do
                {
                New (curr_ptr);
                Writeln (curr_ptr);
                curr_ptr^ := entry;

                Readln (entry);
                };
End
{
-------------------------------------------------------------------
Figure 4-19.    DYNA.PAS        Example of POINTERs and the HEAP
*******************************************************************}
```

The actual array that will contain the names is not explicitly declared anywhere in a VAR statement. Instead, we define the type "name" to be LSTRING type of length 10. Now, we can use "name" as a "type clause" in the VAR declaration.

The pointer is declared with the identifier "curr_ptr." Notice the carat (^); it is the notation for declaring a pointer. The (^) is followed with the type of data to which it will point. In this case, it will be pointing to a variable of type "name," which is really an LSTRING(10). The pointer is only capable of pointing to one name at a time.

The last variable declaration is for the input name. We'll simply call it "entry" and also declare it to be of type "name."

The NEW Procedure

In the body of the program, the WHILE structure will keep the program looping as long as the name that is entered is not "end." The first thing that happens in the loop is a call to the procedure NEW. This

procedure is part of Pascal's runtime memory management utilities. The parameter passed to the procedure is our pointer "curr_ptr." Remember, we've informed the compiler of the type of data the "curr_ptr" points to, namely the LSTRING(10) type.

When we call the NEW procedure, it knows how many bytes to allocate for the dynamic variable. That's because we declared "curr_ptr" as a pointer to an LSTRING(10) type. The compiler knows that an LSTRING(10) will need 11 bytes of memory (one byte for the length). NEW first checks to see if there are enough bytes remaining in the heap to make the allocation. Assuming there is sufficient memory available in the heap, the NEW procedure assigns the next available memory address to the pointer "curr_ptr." For illustration's sake, we print out the value of the pointer right after the NEW procedure.

Then, the name that was just entered is simply assigned to the new variable location on the heap. The notation here involves the (^) again, this time after the identifier. Here it means to assign the value in "entry" to the location in memory specified by the value of the pointer. Finally, we keep the loop going with another READLN statement.

If you just hold down the Enter key, you will see the program run in automatic. It will keep allocating space on the heap for each LSTRING type, even though no characters are entered. If you hold down the Enter key long enough, the program will run out of heap, and an error message will be displayed.

There seems to be one little difficulty though. Every time we call the NEW procedure, it assigns a different value to the pointer, destroying the original value and losing the address of the previous name. What we need to do is establish some kind of continuity throughout the list of names.

Linked List Structure

The *linked list* concept can be used in many situations and with many programming languages. It is used whenever a series of components in a structure are not physically contiguous. The main feature of a linked list is that each component carries a pointer to the next component in the list. The pointer in the last component will take on the unique value NIL. (NIL is a special pointer value that means "not pointing to anything" or "pointer not used.")

With a linked list, the physical arrangement of the list components in the computer's memory is immaterial. They can be scattered all around in memory, as long as each one "knows" where the next one is, and the last one "knows" that it is last. Figure 4-20 shows a graphic representation of a linked list.

Example Using a Linked List

The example program "Namelist," shown in Figure 4-21, illustrates how to create a linked list, and how to print it. Notice the similarities and differences between handling a statically allocated array, like the ones in the previous examples, and handling the dynamically allocated linked list.

Figure 4-21. Example of a linked list

```
{*****************************************************************}
NAMELIST.PAS    Linked List Demonstration Program.
-----------------------------------------------------------------
        This program demonstrates the technique of building a
linked list structure on the HEAP.
-----------------------------------------------------------------
}
Program Namelist (input,output);

Type
        name_fld = lstring(10); { <---- declare the type        }
```

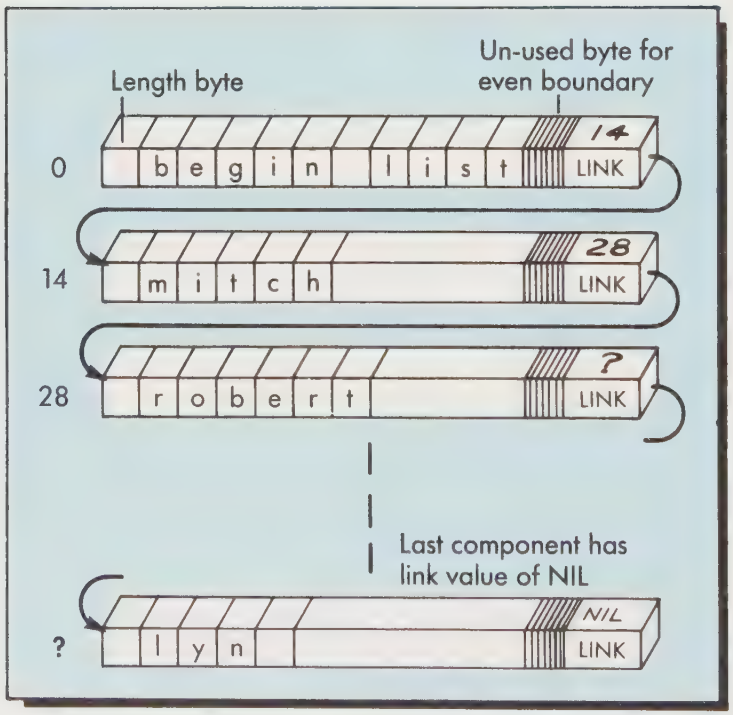


Figure 4-20. Diagram of a linked list

```
        name_ptr^ = name_rec;    { <---- declare the pointer

        name_rec = record        { <---- declare list element
                name    :name_fld;
                next    :name_ptr;
            end;

Var
        first_ptr,               { <---- working pointers
        curr_ptr
                        :name_ptr;
        entry
                        :name_fld;

{------------------------------------------------------------------}
    Main program.  Set up the anchor record for the list.
{------------------------------------------------------------------}

Begin

        New (first_ptr);
        first_ptr^.name := 'Begin list';
        first_ptr^.next := nil;
        curr_ptr := first_ptr;
;
{------------------------------------------------------------------}
        Input the names.  Get a NEW pointer each time, and
display it on the screen.
{------------------------------------------------------------------}
!
        Writeln ('Enter names or end');
        Readln (entry);

        While entry <> 'end' Do
            [
            With curr_ptr^ Do
                [
                New (next);
                curr_ptr := next;
                ];
            Writeln (curr_ptr);
```

```
{-------------------------------------------------------------}
          Set up a new record at the end of the list.
{-------------------------------------------------------------}

          With curr_ptr^ Do
              {
              name := entry;
              next := nil;
              };

          Readln (entry);
          };
{-------------------------------------------------------------}
       Return to the beginning of the list, and display all names.
{-------------------------------------------------------------}

       curr_ptr := first_ptr;

       While curr_ptr <> nil Do
           {
           With curr_ptr^ Do
               {
               Writeln (name);
               curr_ptr := next;
               };
           };

End.
{-------------------------------------------------------------}
Figure 4-21     NAMELIST.PAS    Example of a linked list
*************************************************************}
```

In the Namelist program, we define a data type called "name_ptr." This is our pointer type, and will point to a data type of "name_rec." Note how the (^) is used before the identifier in the type statement. Next, "name_rec" is defined as a RECORD type consisting of two fields. The first field is the name itself. The second field is another pointer, built right into the record. It is this pointer "next" that will point to the next record in the linked list structure. If there is no next record, the value of the pointer should be NIL.

Then we get down to declaring variables. Two pointers will be used in this program. The "first_ptr" will always contain the address of the first record in the list. We're going to use a *list anchor record* as this first record. It's a "dummy" record that will point to the real first record.

(We'll explain more about this later.) The other pointer is called "curr_ptr," and will be our general processing pointer.

The program begins by initializing the anchor record. A new heap address is assigned to "first_ptr" by the NEW procedure. The next two statements illustrate how a field within the newly-allocated record can be accessed. It works just like any other record. The caret (^) is used to indicate that the pointer itself is not being operated upon here. Rather, the contents of the memory location it points to are being operated upon. The field identifier simply follows the "." after the pointer. The anchor record is given the dummy name "Begin list" and a next pointer value of NIL indicating that there are no further records in the list (yet). Finally, "curr_ptr" is assigned the value of "first_ptr," and we are ready to create as many name records as will fit in memory.

Notice how we have simplified the identifer names using the WITH statement. We can refer to the dynamically allocated record using the pointer name followed by the caret (^), just as if it were a RECORD name. In the first WITH statement, a new pointer is assigned to the current (anchor record) record's link, overlaying the old NIL value. Then, that new pointer is assigned to "curr_ptr," and we print it out for fun.

Since we have a new value in "curr_ptr" we can again use the WITH structure to assign the "entry" name to the "name" field in the newly allocated record. Of course, its link pointer is assigned the value NIL to indicate that it is the last record in the list. And we read the next name, etc...

Upon entering "end," the program will leave the WHILE structure and reset the "curr_ptr" to the original "first_ptr" value and begin printing the list. Again using the WITH statement to isolate one record in the list (using the pointer), we can refer only to the field names in the processing. The "curr_ptr" is re-assigned each time through the loop with the value of the link pointer in each record. In this way, the entire list is traversed with relatively few statements.

The SET Type

It is often useful to have a data structure that defines a collection of several logically related components. In other languages such as BASIC, we would probably use an array structure of some sort. In Pascal, we can use the SET type.

The number of members in a SET cannot exceed 255. The SET type and the operators that are associated with SETs are based on the mathematical concept of finite sets. We show SETs and SET operators in Figure 4-22.

SET Operators

There are many mathematical operations that can be performed on SETs, but we will only be concerned with the three SET operations that are supported by Pascal. They are:

SET Union (denoted by "+" operator). For example,

```
set_3 := set_1 + set_2;
```

creates "set_3", a combination of all the members in both "set_1" and "set_2".

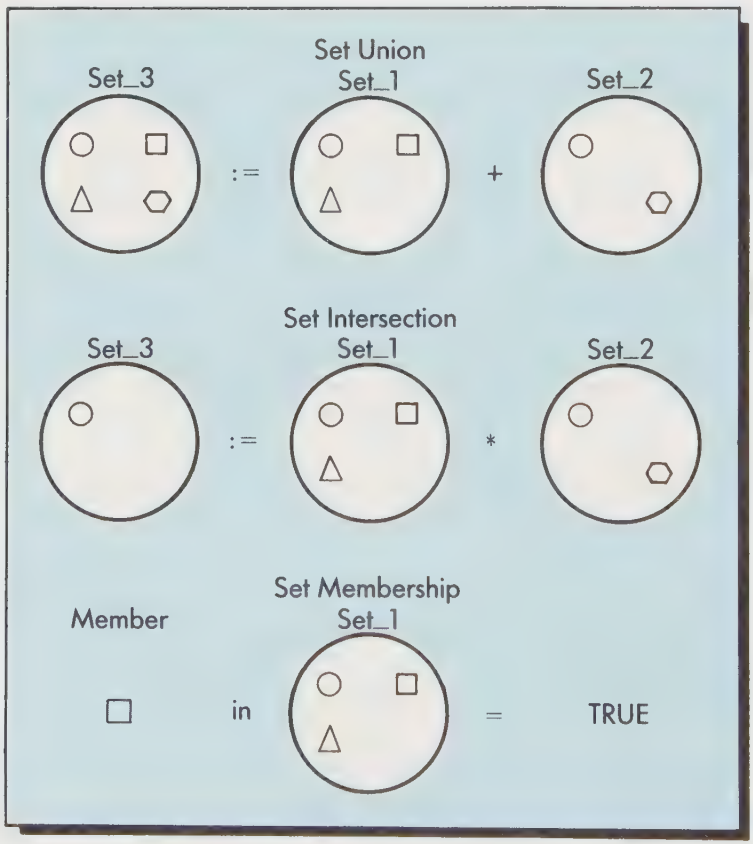


Figure 4-22. Graphic representation of SETs

SET Intersection (denoted by "*" operator). For example,

```
set_3 := set_1 * set_2;
```

creates "set_3", of those members which are common to both "set_1" and "set_2".

SET membership (denoted by "in"). For example,

```
If member in set_1 Then...
```

results in a Boolean evaluation TRUE if "member" is in "set_1," and FALSE if it is not in the set.

A SET variable can also be thought of as an ARRAY OF BOOLEAN values, one for each of the members of the set. If a particular member actually is in the set, then the corresponding component of the array is TRUE. Otherwise, the component indexed by that member is FALSE.

Declaring SET Types

When declaring a SET, it is necessary to specify the *base type* of the members of the set. The base type must be an ordinal type such as:

```
CHAR        <— for characters
(Ø..255)    <— for integers
(x,y,z)     <— for any enumerated type
```

A *constant set* is declared as a list of values, separated by commas, between brackets "[" and "]". For example:

```
vowels    := ['a', 'e', 'i', 'o', 'u'];
```

One particular set can contain no members at all: it's called the "null set." The null set is indicated by empty brackets "[]" and is considered a member of every set.

Example Using a SET

Figure 4-23 is a Pascal program that performs simple calculations similar to a four-function calculator. To perform a calculation, the user simply enters an arithmetic expression consisting of numeric digits and operators.

Figure 4-23. Example using SETs

```
{*************************************************************}
INTCALC.PAS     Integer Calculator Program.
{-------------------------------------------------------------}
     This program operates like a four-function calculator.
Type any integer expression using the digits Ø-9 and the
operators + - * /.  Press ENTER at the end of the expression,
and the program will display the resulting evaluation.
{-------------------------------------------------------------}

Program Intcalc (input,output);

Var
       char_in,                 { character from input          }
       last_op                  { last operation                }
                        char;

       operators                { valid operations      }
                       :set of char;

       digitval,                { value of digit just read      }
       currnum,                 { current value being read      }
       prevnum,                 { previously read value         }
                       :integer;

       operin,                  { true if char_in is an operator }
       digitin,                 { true if char_in is a digit    }

       invalid,                 { true if char_in is invalid    }
                       :boolean;

{-------------------------------------------------------------}
       Initialize the set "operators" to contain only the
four arithmetic operators.
{-------------------------------------------------------------}

Value
       operators := ['+', '-', '*', '/'].

{-------------------------------------------------------------}
     Main program.  Enter the expression and press ENTER.
{-------------------------------------------------------------}

Begin

   Repeat                       { until end of data             }

       Writeln ('Enter calculation');
       currnum := Ø;                    { no previous values    }
```

```
        prevnum := Ø;
        last_op := '+';                 { start with '+'        }

        If  Eof  Then Break;    { done if no more data          }

        While  not (eoln or invalid)  Do
                |
                Read (char_in); { get next character from input }
|
-------------------------------------------------------------------
                Determine type of character.
-------------------------------------------------------------------
|
                digitin := (char_in >= 'Ø')
                                and
                           (char_in <= '9');

                operin := char_in in operators;

                invalid := not ( digitin or operin );
|
-------------------------------------------------------------------
                   Process according to type.
-------------------------------------------------------------------
|
                If  digitin
                    Then [      { convert to integer            }
                        digitval := Ord(char_in) - Ord('Ø');
                        currnum := 1Ø*currnum + digitval;
                        ];

                If  eoln  or  operin    Then
                    Case  last_op  of
                        '+':  prevnum := prevnum + currnum;
                        '-':  prevnum := prevnum - currnum;
                        '*':  prevnum := prevnum * currnum;
                        ' ':  prevnum := prevnum Div currnum;
                              end;

                If    operin
                    Then [      { setup another operator        }
                        currnum := Ø;
                        last_op := char_in;
                        ];

                ];              { while not (eoln or invalid)   }
```

```
{
-------------------------------------------------------------
                    Print the result.
-------------------------------------------------------------

      If  invalid
            Then  Writeln ('Unrecognized character')
            Else  Writeln ('Result is ',prevnum);

-------------------------------------------------------------
                  Move past eoln marker.
-------------------------------------------------------------
}
      If  eoln
            Then  Read (char_in)
            Else  Readln;

   Until  false;              { loop until break executed     }

End.  { intcalc }

-------------------------------------------------------------
Figure 4-22.    INTCALC.PAS     Example using SETs
*************************************************************}
```

Only integers may be used and an operator must be placed between each pair of integers. For example, a calculation might be typed as:

```
15+25*2
```

or as

```
125-86+211/5*18
```

All calculations are done left to right and "/" refers to integer division. The only valid characters are the digits "0" thru "9," and the operators, "+," "-," "*," and "/". Any other character will cause an error message to be displayed.

The SET variable "operators" is used to define the set of valid operators that may be entered. The base type of this set is declared as CHAR, so any character is a potential member. However, in the VALUE section of the program, "operators" is assigned to be the SET declared as:

```
[ "+", "-", "*", "/" ]
```

Thus, although the SET variable operators *may* contain any collection of characters, it *actually* contains just the four characters that indicate the valid operations for the integer calculator.

The BOOLEAN variable "operin" is set TRUE if "char_in" is a valid operator, by the statement:

```
operin := (char_in In operators);
```

This could also have been accomplished with the statement:

```
operin := (char_in = "+")    or
operin := (char_in = "-")    or
operin := (char_in = "*")    or
operin := (char_in = "/");
```

However, using a SET makes it easier to specify and to change if more operators are wanted. Imagine how long the second statement would be if the program allowed 10 different operators instead of only 4.

The heart of the program is a WHILE...DO structure that inputs each character one at a time, until either the whole line is read (EOLN = TRUE), or an invalid character is found. The BOOLEAN variables "digitin" and "operin" are set according to the results of evaluating the set operator IN. Any other character will cause "invalid" to be TRUE which in turn terminates processing of the calculation and displays an error message.

If the character entered is a digit, then the value of "curr_num" is increased. For example, if "2," "3," and "4" are found on three consecutive passes through the loop, then "curr_num" will have the value 2, 23, and 234 respectively at the end of each pass. When an operator is found, then "curr_num" has reached its final value, and it is combined with any previous value "prev_num," using the previous operator, "last-op." The result is saved in "prev_num" and the operator just found is saved as "last-op" until the next number is typed in and "curr_num" will get a new value. When the end of line is reached, the final calculation is done, the loop is terminated, and the result displayed.

The main body of the program is contained a large loop:

```
Repeat
     *    *    *
     If  Eof  Then  Break;
     *    *    *
Until False;
```

This causes the program to repeat indefinitely. The only way to stop the program is by typing the end-of-file character (Ctrl Z) when it asks for another calculation. As always, a Ctrl C can also be used to terminate the program.

Summary

Now that we have finished our discussion of structured data types, you are equipped with many of the tools that you will require to write some wonderful Pascal programs. You can see that Pascal is perfect for any application that requires flexible control over its data. The structured types ARRAY and RECORD will be used quite frequently as we get deeper into programming in Pascal.

Exercises

1. What is the difference between ARRAYS and RECORDS?
2. When is it useful to consider STRINGs and LSTRINGs as arrays?
3. What are the advantages of using dynamic variables and pointers?

Use the following program segment to answer the questions 4, 5, and 6.

```
type
     str50  = lstring(50);
     people = (Bob, Ted, Carol, Alice, Jose)
     group  = set of people;
var
     last_name :array [people] of str50;
     leader    :record
                    who       :people;
                    followers :group;
                end;
```

4. Write the expression for Alice's last name.
5. Write the expression for the leader's name.
6. Write the assignment statement for making everyone except Carol followers of the leader.

Solutions

1. ARRAYS are indexed by integers or enumerated variables; every data element of the array is the same type. RECORDS have separate names for each element; different types of data can be combined in a single record.

2. When individual characters are needed in the middle of a STRING or LSTRING, it is best to consider them as arrays.

3. When using dynamic variables and pointers, the maximum number of items does not have to be known in advance. Also, powerful data structures, such as linked lists, can be built and manipulated.

4. last_name [Alice]

5. last_name [leader.who]

6. leader.followers := [Bob, Ted, Alice, Jose];

5

Functions and Procedures

Concepts

When to use a function or procedure
Pre-declared functions and procedures
Local and external routines
Program segments
Address type variables
Referencing a memory location
The monochrome display buffer
The stack
Parameters and calling conventions
Characters and attributes
Linking assembly language and Pascal
Input/Output ports
The sound chip
Cursor positioning and location

Keywords

FUNCTION, PROCEDURE, RETURN, EXTERNAL, PUBLIC, $INCLUDE, ADR, ADS

A function or procedure can be thought of as a little program in itself. Often referred to as *routines*, they usually have a specific job to do, and are generally part of a larger process that may contain several functions and procedures. We have been using them all along in this book, without making a point of it. For instance, in some of our earliest example programs we used the WRD function to change between INTEGER and WORD types. We also talked about the SQR and SQRT functions for squaring a number and finding its square root. In every chapter we've used the READLN and WRITELN procedures to accomplish input and output. Now we'll explore the general form of functions and procedures.

Function or Procedure

The first thing we have to do is decide where and when to use a function or procedure. If a particular process meets any of the conditions described below, you'd be wise to use a function or procedure.

1. The process is to be applied to several different program variables. (Think of the usefulness of the SQR function.)
2. The process will be required at more than one point in the program. (Remember READLN and WRITELN.)
3. The process will be used by more than one program.

The basic idea is that functions and procedures are specialized program parts that usually perform some specific task, useful in many applications. The SQR function, for example, can be useful in many programs, and perhaps many times within one program.

If the program needs to execute some process at more than one point in the flow, then a function or procedure will provide a more concise program structure. If a process is only used once in a program, then it makes little sense to go through the extra steps to make that process a function or procedure. The program statements for the process should simply be coded into the main program at the point where the process is to be done.

When to Use a Function

The choice between function or procedure is predicated upon one fact: The FUNCTION *returns* a value. Functions are usually involved in assignment statements. An example of this is the TRUNC function, which truncates a REAL type by dropping the fractional part to the right of the decimal point. The resulting value is returned as an INTEGER type, and is temporarily identified by the function name itself. Figure 5-1 is an example of the use of the TRUNC function.

Notice the assignment statement that invokes the TRUNC function. The identifier within the parentheses of the TRUNC function is called a *parameter* of the function. In this case, it is a REAL type that is to be truncated to an INTEGER type. The value resulting from execution of the function is temporarily accessible through the use of the function name itself. It can therefore be used like a variable identifier in the

assignment statement. The truncated value is assigned to the INTEGER variable "whole_part," which is then printed.

Generally, a function does not alter any other data elements. It should simply take the parameter within the parentheses, perform its process, and return the resulting value for further processing by the program. While it is possible for a function to alter other data, this is really a job for a procedure.

Figure 5-1. REAL to INTEGER conversion

```
{*************************************************************
TRUNCER.PAS     TRUNC function demo program
-------------------------------------------------------------
        This program illustrates the use of the TRUNCate
function to convert REAL type data into INTEGER type.
-------------------------------------------------------------}

Program Truncer (input,output);

Var
        real_number     :real;
        whole_part      :integer;

Begin
        Writeln ('Enter a real number');
        Writeln ('The whole part will be returned');
        Readln (real_number);

        While real_number <> 0.0 Do
                {
                whole_part := Trunc (real_number);
                Writeln (whole_part);

                Readln (real_number);
                }:
End.

-------------------------------------------------------------
Figure 5-1      TRUNCER.PAS     REAL to INTEGER Conversion.
*************************************************************}
```

When to Use a Procedure

A PROCEDURE may perform any kind of programming task. It can alter many data elements. The returned value can be accessed directly, rather than only by the name of the function. A procedure can involve the use of other procedures or functions.

Procedures are invoked by the appearance of their identifier in a

Pascal statement. The identifier is then followed by a list of the parameters, enclosed in parentheses. Two procedures we should be very familiar with by now are READLN and WRITELN. The parameter list for these two procedures contains the data elements to be input or output.

Predeclared Functions and Procedures

All the functions and procedures we've encountered so far have been *predeclared*. That is, the nature of their operation has been built into the Pascal compiler. All you have to do is mention their names and the compiler knows what to do. In some cases, the compiler just generates code to be inserted at the point of the *call* to the function or procedure. In other cases, the compiler sets up the *calling sequence* for those predeclared functions and procedures. We will discuss this in more detail later in this chapter.

Local Functions and Procedures

There are other types of functions and procedures which the compiler doesn't know about automatically. They are most commonly declared within a program and used by that program and none other. A function or procedure has a structure that is similar to overall program structure. Remember: We said they were like little programs. Well, that's just the way the compiler treats them. The compiler wants all functions and procedures to be declared before the BEGIN statement in the main program.

Functions and procedures may also be EXTERNAL. But let's not worry about that until we've worked through a simple example using a function and a procedure.

Example Using a Function and Procedure

The program shown in Figure 5-2 uses both a function and a procedure. It should be a useful utility program for your IBM PC, especially if you ever need a quick way to convert numbers from hexadecimal to decimal. (If you are unfamiliar with the hexadecimal numbering system, now would be a good time to read about it in the Appendix.)

The program's name is HEXEDIT, and its task is to *get* and *put* values represented in hexadecimal. "Get" means to take ASCII hexadecimal characters entered from the keyboard or elsewhere and decode them into a 16-bit WORD type. "Put" means to take a 16-bit

WORD type and encode it into ASCII characters, generally to be output to the video display.

We will be using the function and the procedure defined in this program for some more exciting applications later on in this chapter. For now, we'll use this program to study a new function and a new procedure.

Figure 5-2. Hexadecimal editing program

```
{*****************************************************************}
HEXEDIT.PAS     Hexadecimal editing program
-----------------------------------------------------------------
        This program demonstrates the use of a FUNCTION and a
PROCEDURE to perform hexadecimal editing.
-----------------------------------------------------------------

Program Hexedit (input,output);

Var     { for main program }
        hex_string      :lstring(4);
        word_val        :word;
        mode            :char;

-----------------------------------------------------------------
                GETHEX Hex converter function.
-----------------------------------------------------------------
        This function takes a character string, and returns a
16-bit WORD type value.
-----------------------------------------------------------------
{
Function Gethex (Var chars:lstring(4)) : word;

    Const
        hex_chars = '0123456789abcdef';
        hex_size = 16;

    Var
        digits,
        pos             :integer;
        one_char        :char;
        hex_tot,
        hex_val         :word;

    Begin
        hex_tot := 0;
        digits := Ord (chars.len);  { convert length byte to an integer }
```

```
{
-------------------------------------------------------------------
        Convert one character at a time using the table of hex
characters.
-------------------------------------------------------------------
}
        For pos := 1 to digits Do   {loop through each character in lstring }
            [
            one_char := chars [pos];
            hex_val := Wrd (Scaneq (hex_size, one_char, hex_chars, 1));

            If hex_val = hex_size Then
                [
                Writeln ('Invalid character in position ', pos);
                Return;
                ];
{
-------------------------------------------------------------------
        Now, scale the value based upon the character's position
in the string.
-------------------------------------------------------------------
}
            Case (digits - pos) of
                1:      hex_val := hex_val * 16;
                2:      hex_val := hex_val * 256;
                3:      hex_val := hex_val * 4Ø96;
                Otherwise;
                    end;

            hex_tot := hex_tot + hex_val;
            ];
{
-------------------------------------------------------------------
        Finally, assign the accumulated value to the function
identifier name to be returned to the caller.
-------------------------------------------------------------------
}
        Gethex := hex_tot;

        End;                    { end of GETHEX                 }
{
-------------------------------------------------------------------
             PUTHEX    Decimal conversion procedure.
-------------------------------------------------------------------
        This procedure takes a 16-bit WORD type and converts it
into a string of 4 ASCII characters which represent the
hexadecimal value of the WORD.
-------------------------------------------------------------------
}
Procedure Puthex (word_val:word; Var chars:lstring);
```

```
    Const
        hex_chars = 'Ø123456789abcdef';

    Var
        pos             :integer;
        hex_val         :word;

    Begin
{
------------------------------------------------------------------
        Compute the hex character for each position in the 16-bit
word value.
------------------------------------------------------------------

        For pos := 1 to 4 Do
            [
            Case pos of
                1:      [
                        hex_val := word_val Div 4Ø96;
                        word_val := word_val mod 4Ø96;
                        ];
                2:      [
                        hex_val := word_val Div 256;
                        word_val := word_val mod 256;
                        ];
                3:      [
                        hex_val := word_val Div 16;
                        word_val := word_val mod 16;
                        ];
                4:      [
                        hex_val := word_val;
                        word_val := Ø;
                        ];
            end;

            chars [pos] := hex_chars [hex_val + 1];
            ];

        chars [Ø] := chr (4);    { <---- set lstring length.      }

    end;                         { end of PUTHEX                  }

------------------------------------------------------------------
    Main program.  Get the mode GET or PUT from the user.
------------------------------------------------------------------

Begin
    Repeat
        Write ('Get or Put (g/p) ');
        Readln (mode);
```

```
{
-------------------------------------------------------------------
        In the GET mode, input a string of characters and use the
GETHEX function to produce the 16-bit value.
-------------------------------------------------------------------
}
            Case mode of
                'g':    [
                        Writeln ('Enter hex values');
                        Readln (hex_string);
                        While hex_string.len > Ø Do
                            [
                            word_val := Gethex (hex_string);
                            Writeln (word_val);
                            Readln (hex_string);
                            ];
                        ];
{
-------------------------------------------------------------------
        In the PUT mode, input a decimal value, and use the PUTHEX
procedure to convert it into hexadecimal form.
-------------------------------------------------------------------
}
                'p':    [
                        Writeln ('Enter decimal values');
                        Readln (word_val);
                        While word_val > Ø Do
                            [
                            Puthex (word_val, hex_string);
                            Writeln (hex_string);
                            Readln (word_val);
                            ];
                        ];

                Otherwise;
            end;

        Write ('Continue (y/n) ');      Readln (mode);
    Until mode = 'n';

End.
{
-------------------------------------------------------------------
Figure 5-2      HEXEDIT.PAS     Hexadecimal Editing Program.
*******************************************************************}
```

The GETHEX Function

The *get* part of the program shown in Figure 5-2 decodes an LSTRING of up to 4 characters representing a 4-digit hexadecimal value such as 13e, 7fff, or 2c. The get is done by the function GETHEX, which returns a value of type WORD. It has one parameter which is given the formal name "chars." This will be the input to the function. "Chars" is an LSTRING type that will contain the characters representing the hexadecimal value.

The main job in this function occurs in the FOR structure, where each character in the input LSTRING is analyzed.

We use the IBM string function SCANEQ to locate the character in the CONSTant table of hexadecimal characters in the LSTRING named "hex_chars." This position is returned as an INTEGER type, which we immediately change to WORD type using the WRD function. The variable "hex_val" will then contain a value ranging from 0 to 15, unless the character is not found in "hex_chars." In that situation, the value returned would be 16, and an error would be detected.

This test for validity is performed in one IF structure. A message is displayed on the screen indicating that there is an invalid character in a given position. Notice the RETURN statement here. This terminates further execution of the GETHEX function, and returns to the main program, at the point where the function was called.

The next step in the GETHEX function is to scale the hexadecimal digit according to its position from the right in the hexadecimal value. This is done using a CASE structure. The case index is derived from the difference between the total number of digits and the current position being processed.

The last step in the FOR structure is to accumulate the value for the current position by adding it to the total "hex_tot." After all of the characters in the source LSTRING have been processed, the accumulated total in "hex_tot" is assigned to the identifier name "Gethex," and the function returns to the calling program.

Notice that the only value coming out of the GETHEX function is at the end, where it is assigned to the function identifier. Also, there are no other alterations to memory outside of the function itself. This makes GETHEX a *pure* function. It has all of its indexing variables built in, and even has its own LSTRING of hexadecimal characters.

There's one last thing to look at in GETHEX. In the FUNCTION heading we see a VAR declaration for "chars." This is called a *formal reference parameter*. It means that when the function is called, "chars" will contain the **location** of the LSTRING to be processed, not the LSTRING itself. (Later in this chapter we will discuss *calling conventions* for

functions and procedures; that is, how the calling program communicates with the function or procedure.)

The PUTHEX Procedure

The procedure PUTHEX works in an exactly opposite way than GETHEX. PUTHEX takes a 16-bit value and encodes it into 4 ASCII characters in an LSTRING type. For each hexadecimal position, the input value "word_val" is divided by the appropriate position scaler. This integer division truncates the fractional part of the quotient..

The division for each hexadecimal place is performed in a CASE structure within a FOR structure. At the end of the FOR structure, the proper character is located in the "hex_chars" LSTRING and assigned to the output LSTRING "chars" in the corresponding position. Finally, outside of the FOR structure, the length byte (position 0) of "chars" is set to four since there are four characters in the LSTRING.

The HEXEDIT Program

Now that we have declared all of the work to be done by this editing program, we can declare the program flow itself. This is a demonstration program, so we'll allow you to select whether to get or put. Another CASE structure will control the mode until you enter a value of zero. Then you will be given an option whether or not to continue.

Notice how concise this makes the main program. By putting all the "number-crunching" activity in the function and procedure, we can write a main program that is easy to understand. HEXEDIT also demonstrates the fact that functions return just one value, while procedures can alter several values.

Program Segmenting

How could we use the GETHEX and PUTHEX routines in some other program? Our present example has the routines embedded in the code for the main program, but it can be done differently.

There are several ways to go about making Pascal programs available to more than one application. One way of *hooking* programs, functions, and procedures together is called *program segmenting*, and it happens at the source code level.

We can use our word processor or other editor to strip out the GETHEX function and the PUTHEX procedure from our demonstration program HEXEDIT. We'll put each into its own file and call them GETHEX.PAS and PUTHEX.PAS respectively. We can use

either or both of these routines by using one of the the IBM Pascal compiler's *metacommands*. The command we'll want to use is called $INCLUDE. It works quite simply; the general syntax is:

```
{$include:'filename'}
```

Notice that the entire command is enclosed with the braces that normally signify a comment. When the first character of the comment is a dollar sign ($), the compiler interprets the comment as a metacommand. This metacommand tells the compiler to find the file named "filename," read and compile source statements from it until the end of file is reached, and then resume compilation from the original file.

So, we can $INCLUDE the GETHEX and PUTHEX routines in any compilation using the metacommand

```
{$include: 'gethex.pas'}
```

and/or,

```
{$include: 'puthex.pas'}
```

Now that we have such useful tools to edit hexadecimal notation, we can use them to create still other tools.

External Routines

As we mentioned previously, sometimes functions and procedures may be declared EXTERNAL. When they are, only the heading is declared with the word "EXTERNAL" following it. These external functions and procedures reside either in the Pascal library file, PASCAL.LIB, or as distinct object (OBJ) files that can be linked into the final program by the linker. They are called EXTERNAL functions and procedures because they are external to the main program. We will use them quite a bit in our sections on color graphics (in Chapter 7, "Systems of Programs").

IBM Pascal supplies several EXTERNAL routines to provide *low-level* access to the system. Included are procedures to find out the date and time and a function to directly access the PC-DOS.

Example of an EXTERNAL Function

Let's look at a function that the IBM manual almost ignores entirely: the DOSXQQ function. This EXTERNAL function allows limited access to the operating system, PC-DOS.

The operating system is a group of routines that handle all input and output for programs. When a program is running, it usually performs all the data transfers for input/output by *calling* the operating system. DOS calls are inherently low-level and involve the *registers* inside the 8088 chip used by the IBM-PC.

To use any EXTERNAL function or procedure we must tell the compiler about it. We do this by declaring the routine's name and parameters as usual; then, instead of any further declarations or body, we simply use the word "EXTERNAL." For example, the DOSXQQ function is declared with the statements:

```
Function dosxqq (dosnum:Byte; dxreq:Word): Byte; External;
```

There are over 80 DOS calls for the PC-DOS and the first argument indicates which of these you want. The second argument becomes the value of the DX register before calling the DOS. The function returns the value of the AL register after the call. Each DOS call uses these registers of the 8088 chip differently. Some calls also return values in the CX and DX registers. These values can be obtained by using two EXTERNAL variables, CRCXQQ and CRDXQQ, respectively.

One DOS call that may be helpful to you is the keyboard input function. This is DOS call number one and requires no registers except AL in which it returns the byte corresponding to the character typed on the keyboard. This can be accomplished with a statement such as:

```
ch := chr(dosxqq (1,Ø));
```

Notice the CHR function is needed to convert the BYTE value returned by DOSXQQ into a Pascal CHAR type. This might be used to write a program that performs some task on a single keystroke rather than waiting until the ENTER key is pressed.

If you are interested in learning more about DOS calls and their use you should read the IBM PC-DOS manual or *DOS Primer for the IBM PC and XT* by Mitchell Waite, John Angermeyer, and Mark Noble (New York: Plume/Waite, New American Library, 1984), another book in this series.

Since some calls require more registers than AL, CX, and DX you might consider writing your own Pascal function to call the DOS. If you are considering that approach we would highly recommend the Waite Group book *Assembly Language Primer for the IBM PC and XT* by Robert Lafore (New York: Plume/Waite, New American Library, 1984).

ADDRESS Types

There is still a data type that we have not covered. IBM considers the ADDRESS type as a structured type, probably because it can be treated like a RECORD with one or two fields, and can be used like a pointer. The purpose of the ADDRESS type is to provide Pascal with access to any location in the megabyte address space of the IBM PC.

Specifying ADDRESSes

We should explain how addresses are specified in the 8088 microprocessor that is used in the IBM PC. This microprocessor can address one megabyte of memory, which is 1,048,576 bytes. This number, expressed in hexadecimal, is fffff, or five hexadecimal digits. However, the *registers* (internal data processing elements) in the 8088 can only accommodate four hexadecimal digits; that is, numbers up to ffff. Each address must therefore be represented in the 8088's internal registers by two four-digit hexadecimal numbers. The first of these numbers is called the *segment address*, and the second is called the *relative address*.

These two four-digit numbers are combined in an interesting way to yield the necessary five-digit hexadecimal address. The segment address is multiplied by sixteen. This is the same as shifting the number one hexadecimal place to the left. The relative address is then added to the result.

For example, a segment address of b123 and a relative address of 4000 (both in hexadecimal) would be combined this way to yield the absolute address:

```
b123 times 10 = b1230
          + 4000
            b5230
```

This segment:relative form must be used whenever we want to specify an address in the IBM PC.

Using ADDRESS Type Variables

As we will see demonstrated in the next section, the ADDRESS type variable "location" can be used to access the actual memory location specified. This variable is declared in the function's own VAR statement. Variables declared inside the function are only available to that function unless they are declared PUBLIC. In general, that's the way we want it.

There are two varieties of the ADDRESS type variable.

ADR OF (some type) 16-bit relative offset.
ADS OF (some type) 16-bit segment address, plus
16-bit relative offset.

The ADR type provides only the relative offset part of the address, using whatever segment is currently in use. The ADS type provides both the segment and the relative parts of the actual memory location.

This type of variable — ADR or ADS — can be thought of as a pointer. It "points" to an actual memory location just like pointers do, and it may be used to *reference* a memory location. (By "reference" we mean to get the value that is stored at the memory location.) The compiler will need to know what type of data is stored at the address being referenced. That's why the full type declaration might look like this.

```
Var
    start     :adr of array[1..100] of integer;
    current   :adr of integer;
    buffer    :ads of char;
```

As you can see, there are as many different address declarations as there are types.

The ADS type can be treated just as if it were a record. The segment and relative parts of the address can be denoted with the record notation using a period followed by the field name.

Developing PEEK and POKE for Pascal

Two tools familiar to most BASIC programmers, but painfully absent from the Pascal language, are PEEK and POKE. These BASIC routines directly access any memory location by its address. Now we have everything we need to create Pascal equivalents of PEEK and POKE.

The PEEK Function

The PEEK operation allows us to literally "peek" at a given byte in memory to see what binary value is stored there. This value could be a character, or part of some numeric data, or even part of a program instruction. Generally, if you know where to peek, you will know what to expect there. Usually, we will want to assign the byte value that we peek

at to some variable identifier within our program. That's why PEEK should be a function, since it will return a value.

Here is how our PEEK function might be used in a program:

There are two parameters for PEEK. The first is the 16-bit segment address and the second is the 16-bit relative offset part of the address to peek into. Both of these must be supplied by the calling program. The function will return an 8-bit value. Figure 5-3 shows the source code for the PEEK function.

Figure 5-3. Memory PEEK function

```
{*****************************************************************
PEEK.PAS        Memory peek function.
-----------------------------------------------------------------
        This function simulates the operation of the BASIC PEEK
command.  It returns the 8-bit byte value of the memory location
specified by the segment and relative address passed as parameters.
-----------------------------------------------------------------
}
Function Peek (seg, rel :word)  :byte;

Var
        location                { <---- address type variable   }
                        :ads of byte;

Begin
        location.s := seg;
        location.r := rel;
        Peek := location^;
end;
{
-----------------------------------------------------------------
Figure 5-3      PEEK.PAS        Memory PEEK Function.
*****************************************************************}
```

The function heading includes all the information about the parameters that are passed by the caller. In this case, there are two parameters being passed: the segment and relative parts of the address. These are both 16-bit quantities, and are declared in the heading with the type WORD. Notice how the type of the result is declared with the type clause BYTE, right after the heading.

For the PEEK function, we have chosen to use the ADS type, and provide both the segment and relative parts as parameters. This allows PEEK to access any part of memory.

The variable "location" is going to belong to the PEEK function, for its use alone. We have declared "location" as type ADS OF BYTE. This tells the compiler that "location" will be the full 2-word address of a byte somewhere in memory.

In Figure 5-3, "location.s" and "location.r" are simply assigned whatever 16-bit values were passed by the caller. Again, using pointer notation, the memory location specified by the ADS type is referenced through the use of the caret (^) following the identifier name. The 8-bit value residing at the specified memory location is assigned to the function identifier PEEK.

The POKE Procedure

The POKE procedure is rather similar to PEEK. It uses the same principle for addressing the memory: the address type variable. But instead of *getting* the value at the location and returning it as PEEK does, the POKE procedure will *put* some value into the location. This value is also supplied by the caller in the form of an 8-bit quantity as an additional parameter. Figure 5-4 shows the source statements for POKE.

Figure 5-4. Memory POKE procedure

```
{*****************************************************************
POKE.PAS        Memory poke procedure.
-----------------------------------------------------------------
        This procedure is used to simulate the operation of the
BASIC POKE command.  The value contained in "data" is placed in
the memory location specified by the segment and relative
address.
-----------------------------------------------------------------}

Procedure Poke (seg, rel :word;  data:byte);

Var
        location                { <---- address type variable   }
                        :ads of byte;

Begin
        location.s := seg;
        location.r := rel;
        location^:= data;
end;
{
-----------------------------------------------------------------
Figure 5-4      POKE.PAS        Memory POKE Procedure.
*****************************************************************}
```

The operation of the procedure is again quite simple. An address type variable called "location" is used to point to the memory location. Then, using the caret (^) symbol, the value of "data" is assigned to the memory location being pointed to by "location." Note that this "location" is a different variable entirely from the variable that appears in the PEEK function. The fact that they have the same name doesn't mean a thing, since each is used in a different function or procedure.

Using the PEEK Function

Great! Now that we have PEEK and POKE set up, let's put together a demonstration program. It must be able to both PEEK and POKE any memory location with an 8-bit value. All input and output will be in hexadecimal. We'll use a little of everything we've learned so far. The PEEKPOKE program in Figure 5-5 demonstrates the ability to PEEK and POKE with Pascal.

Figure 5-5. PEEK and POKE demonstration program

```
{*****************************************************************
PEEKPOKE.PAS    PEEK and POKE demonstration program.
-----------------------------------------------------------------
        This program is used to demonstrate the operation of the
PEEK function and the POKE procedure.  It also uses the routines
GETHEX and PUTHEX to do the decimal to hexadecimal conversion.
-----------------------------------------------------------------}

Program Peekpoke (input,output);

Var
        mode,                   { mode indicator                }
        seg_str,                { string for segment address    }
        rel_str,                { string for relative address   }
        hex_str,                { hexadecimal string            }
                        :lstring(4);

        seg_wrd,                { segment address               }
        rel_wrd,                { relative address              }
                        :word;

        data            :byte;

{-----------------------------------------------------------------
     Include the source code segments for all the routines.
-----------------------------------------------------------------}

{$include:'gethex.pas'}
{$include:'puthex.pas'}
```

```
{$include:'peek.pas'}
{$include:'poke.pas'}
{
{-------------------------------------------------------------------}
{     Main program.  Demonstrate both PEEK and POKE modes.          }
{-------------------------------------------------------------------}

Begin
    Repeat
        Write ('Enter peek or poke ');
        Readln (mode);
{
{-------------------------------------------------------------------}
{                  Get the segment address.                         }
{-------------------------------------------------------------------}
{
        Write ('Enter segment address in hex ');
        Readln (seg_str);_
        seg_wrd := Gethex (seg_str);
{
{-------------------------------------------------------------------}
{        PEEK mode.  Get the address into which to PEEK.            }
{-------------------------------------------------------------------}
{
        If mode = 'peek' then
                {
                Writeln ('Enter relative offset address in hex');
                Readln (rel_str);

                While rel_str.len > Ø do
                        {
                        rel_wrd := Gethex (rel_str);

                        data := Peek (seg_wrd, rel_wrd);
                        Puthex (data, hex_str);
                        Writeln (hex_str);

                        Readln (rel_str);
                        }.
                }.

{-------------------------------------------------------------------}
{           POKE mode.  Get the address and the data.               }
{-------------------------------------------------------------------}

        If mode = 'poke' then
                {
                Writeln ('Enter relative offset address in hex');
                Readln (rel_str);
```

```
            While rel_str.len > Ø do
                    [
                    rel_wrd := Gethex (rel_str);
                    Write ('Data ');
                    Readln (hex_str);
                    data := Gethex (hex_str);
                    Poke (seg_wrd, rel_wrd, data);

                    Readln (rel_str);
                    ];
            ];

        Write ('Continue (y/n) ');
        Readln (mode);

    Until mode = 'n';
End.
```

```
-------------------------------------------------------------------
Figure 5-5.     PEEKPOKE.PAS    PEEK and POKE Demo Program.
*******************************************************************
```

The whole program is built inside a REPEAT structure. The structure has two main sections, one for PEEK and one for POKE. We'll use several LSTRING types to contain the hexadecimal addresses and data. Two WORD types will be used to provide the binary memory addresses.

Each section is inside a WHILE structure. Execution will remain here as long as relative addresses are entered. If the ENTER key is pressed without entering an address, the length of the lstring will be 0; a simple way to exit the WHILE.

(By the way... can you see another potential application for a function or procedure within this demonstration program? It's something that happens in both PEEK and POKE.)

Well, now that we have the PEEK and POKE features available, let's take a look at what we can use them for. There are several memory locations that would be interesting to peek into. We'll always write the segmented address in the form "seg:rel."

Equipment Flag

The bytes at 40:10 and 40:11 contain the IBM equipment flag. This is set up during power on, and used by the DOS to determine the configuration of the system. Now that you can PEEK with Pascal, you may want to include this ability in one of your own programs. Here's what the equipment flag looks like.

```
40:10 bit 0     diskette flag
                  0 – no diskettes
                  1 – diskettes on the system
          1     not used
          2-3   on board RAM size
                  00 – 16K
                  10 – 32K
                  01 – 48K
                  11 – 64K
          4-5   initial video mode
                  00 – not used
                  10 – 40x25 color
                  01 – 80x25 color
                  11 – 80x25 monochrome
          6-7   number of disk drives on the system
                (only if bit 0 is on)
                  00 – 1 disk
                  10 – 2 disks
                  01 – 3 disks
                  11 – 4 disks

40:11 bit 0     not used
          1-3   number of RS232 cards attached
                (0 through 7)
          4     set if game I/O attached
          5     not used
          6-7   number of printers attached (0 through 3)
```

Remember that we can only peek at one byte at a time with the function we have now. Can you figure out how to modify it to display more than one byte?

Other PEEKs

Here are a few other interesting places to peek. Some of them can also be POKEd with data to change the operation of the system, but such "back door" approaches can be dangerous as we'll see.

40:13	memory size in K bytes
40:15	I/O RAM size in K bytes
40:49	current CRT mode 00 – 40x25 black & white 01 – 40x25 color 02 – 80x25 black & white 03 – 80x25 color 04 – 320x200 color 05 – 320x200 black & white 06 – 640x200 black & white
40:4a	number of CRT columns
40:6c – 40:6d	low word of timer count
40:6e – 40:6f	high word of timer count

Using the POKE Procedure

This is where it starts to be fun. Using the PEEKPOKE demonstration program we can also POKE values into memory. We have to be careful doing this, since we could *crash* the system by poking somewhere we shouldn't. For example, if we POKE the CRT mode byte at location 40:49 with a value of 06, the system will go into the 640x200 high resolution graphics mode, and unless our program is ready to operate in that mode, we'll lose the picture.

The Monochrome Display Buffer

One area that is really fun to poke around in is the *Monochrome Display Buffer*. Everything that appears on the monochrome display is actually contained in this buffer, which resides in the system memory space. There is also a buffer for the color graphics adapter which we'll discuss soon. For now, let's play around with the monochrome display.

Figure 5-6 shows that the monochrome display consists of 25 lines of 80 columns each. This makes a total of 2,000 character positions. Each one of these character positions occupies 2 bytes of the display buffer.

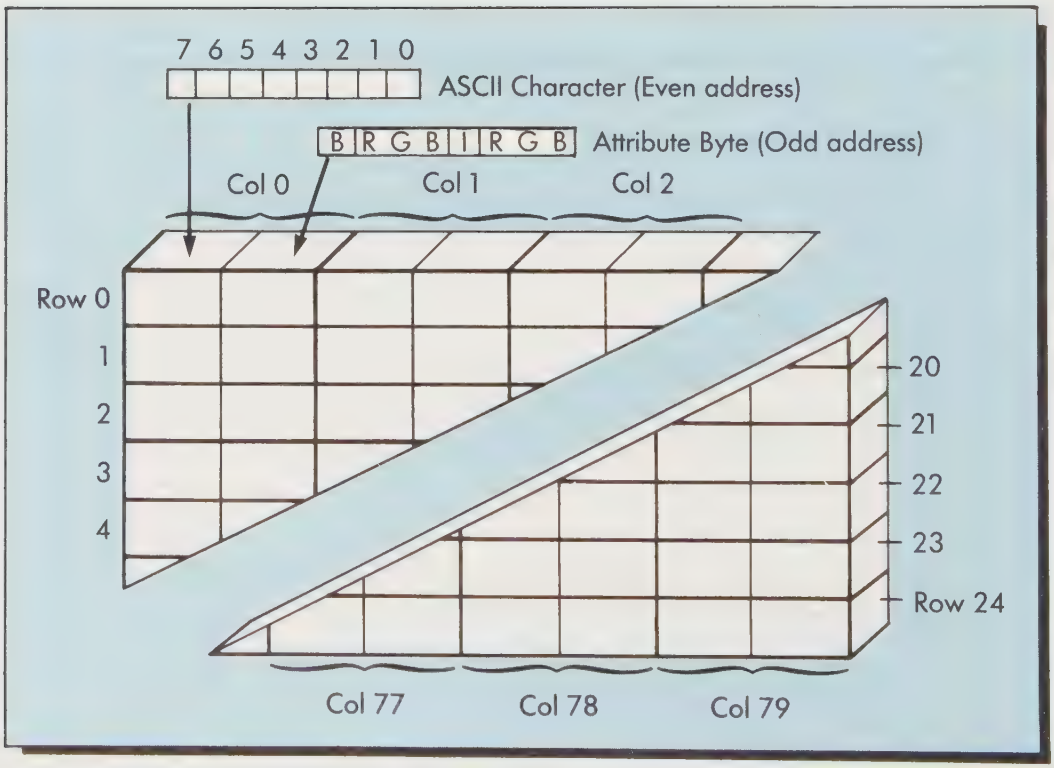


Figure 5-6. Monochrome display buffer

Attributes

The first byte is at an even address and contains the ASCII character itself. The second byte, located at the next memory location above the character, contains the *attribute*. Each character position on the screen can be assigned attributes such as "blinking," "reverse video," and "underline," just by setting the proper value into the attribute byte. The following list shows the assignments of the bits in each attribute byte:

```
Byte 0 – ASCII character (even address).

Byte 1 – attribute byte (odd address).
         bits 0-2 foreground video
                  000 – non-display
                  001 – underline
                  111 – normal (white on black)

         bit 3    intensity
                  0 – low intensity
                  1 – high intensity

         bits 4-6 background video
                  000 – select foreground
                  111 – reverse video

         bit 7    blink
                  0 – no blink
                  1 – blink
```

The Buffer in Memory

The monochrome display buffer is located in the system memory at address B0000 hexadecimal, which can be represented in segment:relative form as b000:0000. Since there are 2,000 character positions and 2,000 attribute bytes, the total display buffer size is 4,000 (decimal) bytes. All of the even addresses contain characters; all of the odd addresses contain attribute bytes.

The display buffer is really just another area of system memory that has been set aside for the sole purpose of containing the information that is being displayed on the screen. The CRT controller constantly scans the

display buffer, and generates the appropriate dot matrix pattern in the location specified. Figure 5-7 shows the display buffer in the system memory space.

Accessing the Buffer

Normally, Pascal's input and output are handled by calls to the resident DOS routines. These take care of putting characters on the screen, as for example, a Pascal program uses WRITELN. There is another way to output data directly from a Pascal program, and have it appear on the screen. We can use the POKE procedure to just "stuff" the data into the display buffer.

To experiment with this, we'll use the PEEKPOKE program and try poking around in the display buffer. Let's cause a "happy face" to appear in the middle of the screen. Remember, there are 2,000 character positions on the screen, so the one thousandth position is the one we are after to get the middle of the screen. But, each position consists of a character byte and an attribute byte, so the one thousandth position is actually relative bytes 1,998 and 1,999 (since we start counting with 0).

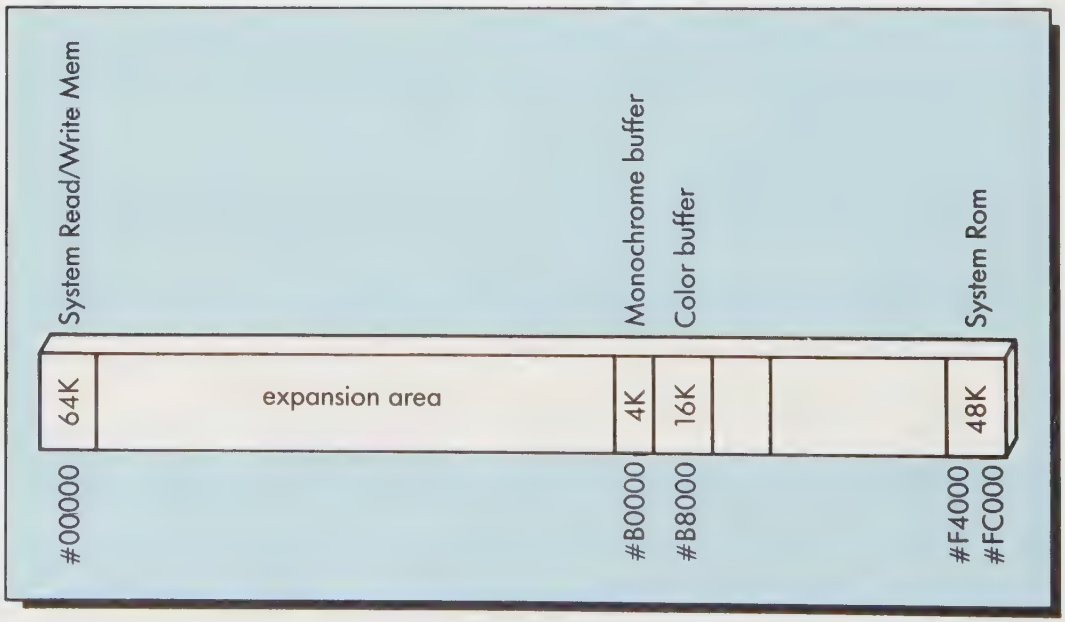


Figure 5-7. Display buffer in system memory

The character is stored in relative byte 1,998, and the attribute in byte 1,999; these are 7ce and 7cf in hexadecimal.

Here's what to do when you run PEEKPOKE.

```
A>peekpoke
Enter peek or poke --->poke
Enter segment address in hex --->bØØØ
Enter relative offset address in hex
7ce
Data --->2
```

At this point, you will see a "happy face" appear in the middle of the screen.

Let's do it once more and notice what happens.

```
7ce
Data --->2
```

The first "happy face" is *scrolling* up the screen with each carriage return as we enter the addresses and data values to the PEEKPOKE program. This is because the CRT controller still is in charge of the display buffer. Since the Pascal READLN and WRITELN procedures use the resident DOS routines to input and output, the scrolling is automatic. Each pass through the PEEKPOKE program will cause the entire screen to scroll up two lines (unless you've just cleared the screen with Ctrl Alt Del, in which case the cursor will not yet be at the bottom of the screen).

Let's see if we can be clever and locate the first "happy face." We'll actually locate its attribute byte, and make it blink in high intensity. The "happy face" has scrolled up 4 lines, each line containing 80 characters and 80 attribute bytes. That puts it 640 bytes below its original location in the display buffer. Actually, we want the attribute byte just above the "happy face" character, so that's only 639 bytes below the original location. Subtracting 639 from 1998 gives 1359. Converting to hexadecimal for PEEKPOKE yields:

```
54f
Data --->8f
```

There, we found the correct attribute byte, and made that little face blink.

Faster Buffer Access

Poking one character at a time into the buffer isn't much fun is it? Let's use some of the skills we've learned so far to experiment more. The next program will fill half of the display buffer with whatever character and attribute we enter from the keyboard. The program shown in Figure 5-8 will give you an idea of how fast Pascal operates.

Figure 5-8. Monochrome display demonstration program

```
{*************************************************************}
MONOFILL.PAS    Monochrome display demo
-------------------------------------------------------------
        This program fills about half of the monochrome display
buffer with the character and attribute values entered by the
user.
-------------------------------------------------------------
}
Program Monofill (input,output);

Var
    char_str,
    attr_str    :lstring(2);

    char_byte,
    attr_byte   :byte;

    seg_addr,
    rel_addr    :word;

Value
    seg_addr := #b000;          { <---- base address for display }

-------------------------------------------------------------
    Include the GETHEX and POKE routines in the compilation.
-------------------------------------------------------------
}
{$include:      'gethex.pas'}
{$include:      'poke.pas'}
}
-------------------------------------------------------------
        Main program.  Input the character and attribute from the
user and fill half of the display buffer.
-------------------------------------------------------------

Begin
    Writeln ('Enter character and attribute  (CCAA)');
```

```
    Readln (char_str, attr_str);

    While char_str.len > Ø do
        |
        char_byte := Gethex (char_str);
        attr_byte := Gethex (attr_str);

-------------------------------------------------------------------
        POKE the character byte into all the even memory
locations, and the attribute byte into all the odd locations.
-------------------------------------------------------------------

        For rel_addr := Ø to 1999 do
            If Odd (rel_addr)
                Then Poke (seg_addr, rel_addr, attr_byte)
                Else Poke (seg_addr, rel_addr, char_byte);

        Readln (char_str, attr_str);


End.

-------------------------------------------------------------------
Figure 5-8.      MONOFILL.PAS    Monochrome Display Demo.
*******************************************************************
```

For the purpose of this demonstration, we'll enter the hexadecimal value for the character, followed immediately by the hexadecimal value for the attribute. This will represent a four-character group that looks like this:

CCAA	where, CC – character value in hex AA – attribute value in hex

You can use any hexadecimal value for the character part, from 00 to ff, although not all of them are displayable on the IBM monochrome display. There is of course the entire alternate character set which contains some graphics segments that can be used for drawing such designs as boxes.

The attribute byte will only be meaningful if it is entered as described earlier. Here are some easy values to try.

Character attribute	Video level Low	High	Rev
normal character display	07	0F	70
underlined character	01	09	—
blinking character	87	8F	F0
blinking and underlined	81	89	—

Conflict with the CRT Controller

If you try the monochrome display demonstration program on your PC, then you'll realize that we still have the same old problem: Every time we use Pascal's standard input/output (I/O) routines READLN and WRITELN, the CRT controller scrolls up the entire screen. If this didn't happen, we could easily put fancy screens together for more interactive data entry. However, with the standard routines being used, we get kind of a teletype feeling as each line is scrolled up the page.

To avoid this problem we need to send information to the CRT controller **directly** from a Pascal program, rather than using Pascal's standard I/O routines. Then, we could control both scrolling and cursor position. But the CRT controller takes its orders through an I/O *port*. We cannot access this port using address types from Pascal. That's because ports are not memory locations; they are special I/O registers that can only be accessed using the machine language IN and OUT instruction.

Outside Help

IBM PC Pascal just doesn't include any routines for getting to the CRT controller directly. What we need is a machine language IN and OUT instruction that we can somehow bring into our Pascal programs. There is a way of doing this, using routines that are written in IBM PC assembly language. But first we need some more background for this to make sense.

The Parameters

In order to understand how we can use routines written in assembly language to enhance our Pascal programs, we must examine how Pascal functions and procedures are implemented. One of the major reasons for using a function or procedure is to be able to apply a given process to many different sets of data. This is why functions and procedures usually have a *parameter list* enclosed in parentheses as part of the declaration. An example of this is shown in Figure 5-9.

Figure 5-9. Minimum value function

```
{*****************************************************************}
MINVAL.PAS      Minimum value function
-----------------------------------------------------------------
        This function takes two integer values as parameters.
It will return the smaller value to the caller.
-----------------------------------------------------------------
}
Function Minval (first, second :integer)     :integer;

Begin
        If first > second
                Then Minval := second
                Else Minval := first;
End;
{
-----------------------------------------------------------------
Figure 5-9.     MINVAL.PAS      Minimum Value Function.
*****************************************************************}
```

In the function declaration in Figure 5-9, there are two parameters: "first" and "second." These are not actually variables, but **represent** the variables that will be used during the execution of the function. These are known as *formal parameters*, and allow the compiler to prepare for the actual values that will be present during execution.

Of course we have to tell the compiler what type the variables will be when they do come along. Parameters can be declared in groups, as shown. Both "first" and "second" are declared as INTEGER types. Since functions return values, we must also tell the compiler the type of the returned value. In our example, the returned value will also be of INTEGER type.

The reason we must declare all this to the compiler is that special code will be generated to connect the function or procedure with the calling program.

Using the Stack

This exchange of data between caller and callee will all take place on the *system stack*. This is an area of memory that is automatically set aside just for this purpose. It is called a stack because it can be visualized that way.

An 8088 CPU register keeps track of the location of the top of the stack. When a calling program wishes to use the stack to communicate with a function or procedure, it must push the necessary data onto the stack before calling. Then of course the function or procedure which is called must pop the data off of the stack, and use it appropriately. This is demonstrated in Figure 5-10.

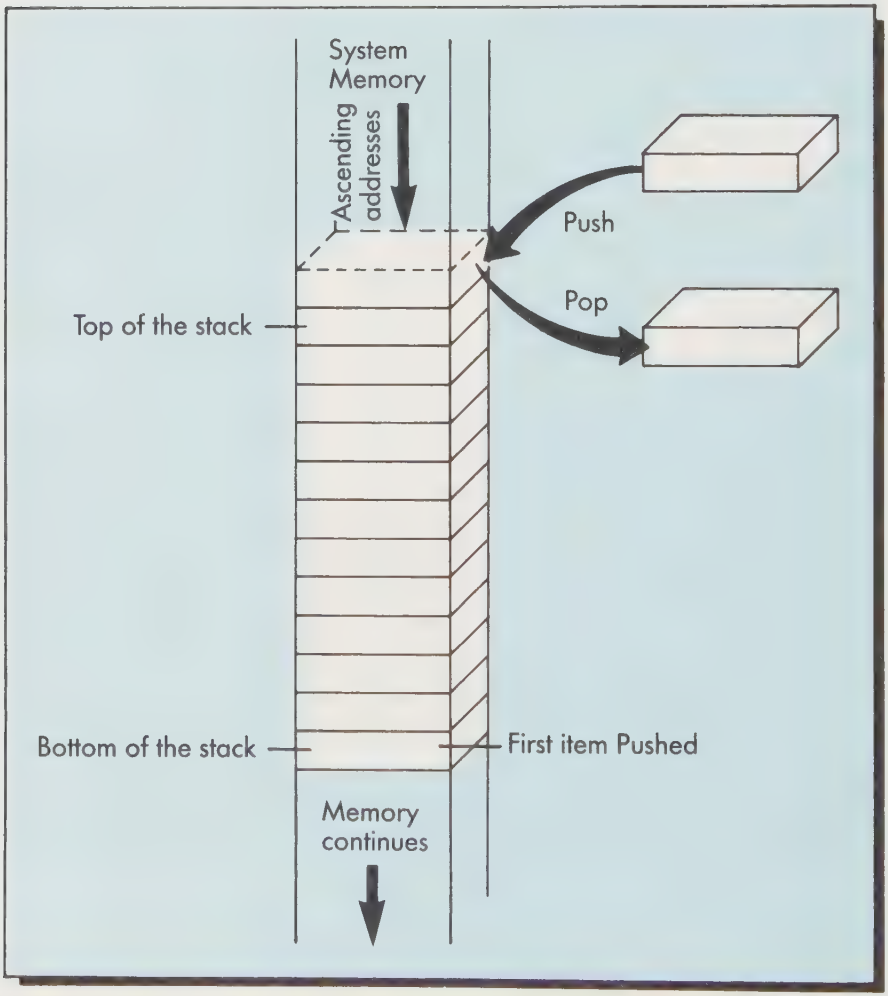


Figure 5-10. Using the stack for communicating

Remember, the stack is just another area of memory, so all we have here is a series of 8-bit values, unless we know what to expect. That's why the function and procedure declarations so thoroughly specify the data types. Figure 5-11 depicts what the stack would look like during the function call to MINVAL, the minimum value function we saw in Figure 5-9.

Also placed on the stack is the *return address* of the calling program. This will be the memory address within the calling program at which execution will be resumed when the function is completed.

Value Parameters

Since there are only two parameters for the minimum value function, and they are both integers, we've declared them as *value parameters*. This means that their identifier names appear in the function heading without the keyword VAR or CONST preceding them. The compiler will generate code to put the value of the actual variables represented by "first" and "second" right on the stack as shown. Since these are INTEGER types, each will occupy two bytes of the stack, four bytes in all for them both.

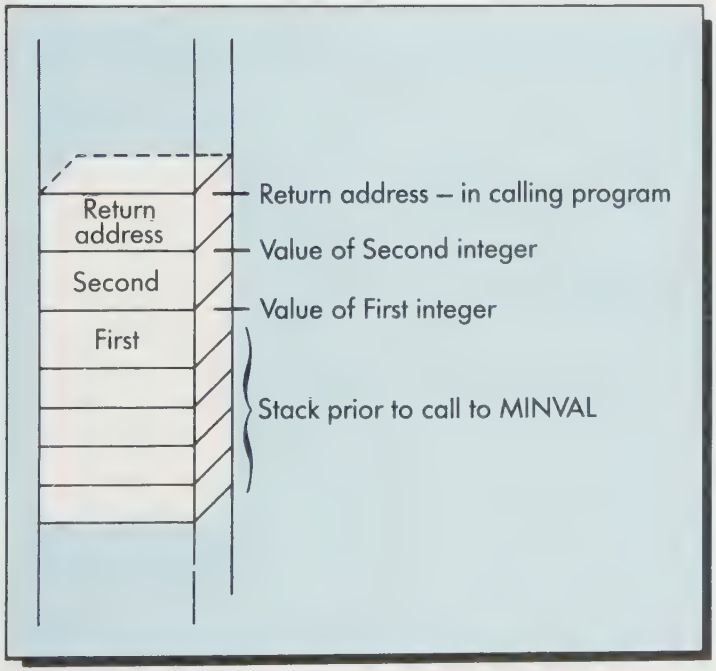


Figure 5-11. Stack during call to MINVAL

This is acceptable when we're only dealing with a couple of INTEGER types, but not good if we want to work on a large data structure of some kind. It also doesn't work when we want to alter one of the parameters during the function or procedure. Since value parameters are only replicas of the parameters that appeared in the call, altering them would not affect the actual variable itself. When either of these conditions occurs, we must declare the parameter differently, by using VAR.

Reference Parameters

Let's suppose we want a function to examine an array of 100 REAL type variables, and return the smallest value. We wouldn't want to put a duplicate of the whole array on the stack; it would take too much memory. (A REAL type requires 4 bytes of memory. So, 100 REALs would take 400 bytes of the stack.) Instead, we will declare the function using a *reference parameter* and the keyword VAR right in the function heading.

The VAR or CONST keywords tell the compiler that we are not expecting the actual values to appear on the stack. Rather, we will be expecting the memory address of the variable. In this case, it will be the address of the ARRAY OF REAL variables that is to be searched for its lowest value. Shown in Figure 5-12 is the program TESTMIN, which includes a function called "Min_array" that does just what we've been talking about.

Figure 5-12. Finding the minimum value in an array

```
{*****************************************************************
TESTMIN.PAS     Find minimum value in array
-----------------------------------------------------------------
        This program illustrates how to find the smallest value in
an array of REAL types.
-----------------------------------------------------------------}

Program Testmin;                { <---- no input or output      }

Type
        reals = array [1..100] of real;

Var
        scores  :reals;         { <---- array of scores         }
        min     :real;
```

```
{-------------------------------------------------------------------}
          MIN_ARRAY function finds smallest value.
{-------------------------------------------------------------------}
;
Function Min_array (Var test_scores :reals) :real;

    Var
        test    :real;
        index   :integer;

    Begin
        test := test_scores [1];
        For index := 2 to 100 do
            If test_scores [index] < test
                Then test := test_scores [index];
        Min_array := test;
    End;
{
{-------------------------------------------------------------------}
                  Main program.
{-------------------------------------------------------------------}
{
Begin
        min := Min_array (scores);
End.
.
{-------------------------------------------------------------------}
Figure 5-12     TESTMIN.PAS     Find Minimum Value in an Array.
*******************************************************************{
```

One lesson here is the way that the structured type is declared in the function heading. If we try to say it all in the heading, (*i.e.*, "array[1..100] of real") the compiler issues an error. But, if we define the type in a TYPE statement, we can use the new type identifier in the function heading. Also, the keyword VAR tells the compiler this is not a value, but a reference parameter. As illustrated in Figure 5-13, on the following page, only the memory address of the array will be pushed onto the stack, not the array itself.

Choosing Parameters

Value and reference are the two most common kinds of parameters. You will need to decide which to use.

Value parameters provide an important advantage. When calling the function or procedure, value parameters can be a calculation of some sort instead of a variable name. When a calculation is used for a value parameter, the stack contains the result of the calculation. For example:

```
limit = minval (3*a+b, 2*(a+b))
```

Here the two calculations will be performed, and the results pushed on to the stack. "Minval" will then do its thing and the smaller of the two results will be returned and become the value of "limit."

In contrast, reference parameters must be variables because there must be some portion of memory reserved for the value of the argument.

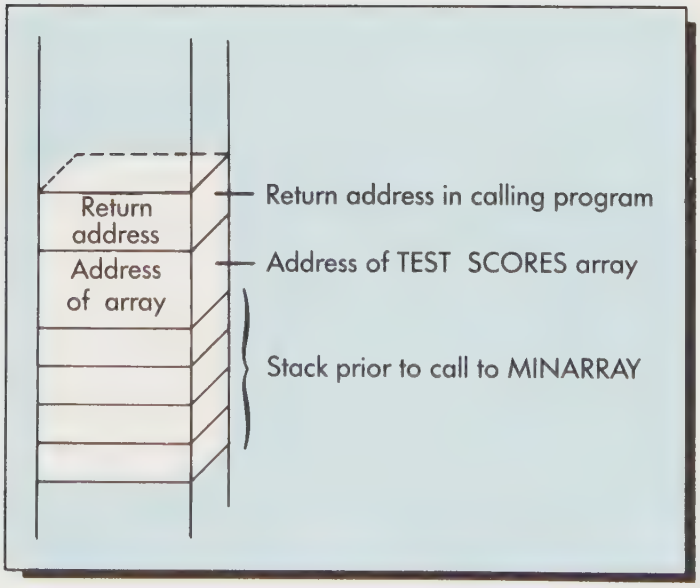


Figure 5-13. Stack during call to MINARRAY

So, to summarize these parameters:

Value parameter
- Passes a replica of the data
- Inefficiently uses the stack
- Cannot be altered
- Can be called with a calculation

Reference parameter
- Sends only the address of the data
- Efficient with data structures
- Can be altered unless CONST
- Must be called with a VAR

Linking Assembly Routines to Pascal

Remember the problem we had with not being able to do IN and OUT with Pascal? But in assembly language it's easy to perform IN and OUT...they are actual CPU instructions. The IN instruction will input from the specified input-output port, and store the 8-bit value found there in a CPU register. The OUT instruction will output the contents of an 8-bit CPU register to the specified port. If we could use them, such routines in Pascal would take the following form.

```
Function Portin (port_number:word): byte;

Procedure Portout (port_number:word; data:byte);
```

PORTIN would be a function because it returns a value to the calling program, and PORTOUT would be a procedure. What we need then is some way of hooking up an assembly language IN and OUT instruction to our Pascal program. The solution is to write the routines using the IBM Macro Assembler, and then link them into an executable program, the EXE file, using the IBM linker program.

External Declaration

Assembly language routines are not declared as part of the source code. Instead, only their headings are declared, with the special keyword EXTERNAL following them. This tells the Pascal compiler that the

function or procedure will be supplied at link time. All of the information necessary to the compiler is present in the heading. Figure 5-14 is a listing of the declarations for the PORTIN and PORTOUT external routines.

Figure 5-14. External port routine declarations

```
{*************************************************************}
PORTIO.PAS      Pascal port I/O external declarations
-------------------------------------------------------------
        This source code segment contains the declarations for the
routines PORTIN and PORTOUT, which are both EXTERNAL.
-------------------------------------------------------------

Function Portin (port_addr:word) :byte;          external;

Procedure Portout (port_addr:word; data:byte);   external;

-------------------------------------------------------------
Figure 5-14     PORTIO.PAS      External Declarations for PORT I/O.
*************************************************************}
```

The PORTIN process will be achieved with a function, since it will return a value, and not alter any other data. PORTIN will have one value parameter "port_addr" of WORD type, the input/output (I/O) port address. It will return the 8-bit value from the port specified. PORTOUT will be a procedure. It does not return a value, but it does alter the contents of an I/O port address. Again, the value parameter "port_addr" specifies the I/O port to receive the output. A second value parameter, "data," contains the 8-bit value to be output to the port.

Pascal IN and OUT

To illustrate how these new external routines can be used, we will use an example similar to the PEEKPOKE program developed earlier in this chapter. We'll call this program PINPOUT for "portin/portout"; its listing is shown in Figure 5-15.

This is almost a carbon copy of the PEEKPOKE demonstration program (Figure 5-5). We have done away with the references to segmented addresses and have replaced them with references to "portin" and "portout." PINPOUT works the same as PEEKPOKE. We can select whether to perform IN or OUT, and then we enter an input/output port address in hexadecimal digits. If we're doing IN, the PORTIN function is executed, returning the value from the port. This is displayed on the screen. If we're doing OUT, the program will ask us to enter the data to

be output to the port. Remember, the data can only be 8-bits, or two hexadecimal digits.

Figure 5-15. PORTIN and PORTOUT demonstration program

```
{*****************************************************************
PINPOUT.PAS     PORTIN and PORTOUT Demonstration Program.
-----------------------------------------------------------------
        This program demonstrates the use of the routines PORTIN
and PORTOUT.  It also uses the routines GETHEX and PUTHEX to get
the port numbers and data from the user.
-----------------------------------------------------------------
}
Program Pinpout (input,output);

Var
        mode,
        port_str,
        hex_str,
                        :lstring(4);

        port_wrd
                        :word;

        data
                        :byte;

-----------------------------------------------------------------
     Include the source code segments for all the routines.
-----------------------------------------------------------------
}
{$include:'gethex.pas'
{$include:'puthex.pas'
{$include:'portio.pas'

-----------------------------------------------------------------
                Main program   Get the mode.
-----------------------------------------------------------------
}
Begin
    Repeat
        Write ('Enter in or out ');
        Readln (mode);

-----------------------------------------------------------------
PORTIN mode.  Get the port number, and then display its contents.
-----------------------------------------------------------------

        If mode = 'in' then
```

```
                [
                Writeln ('Enter port number in hex');
                Readln (port_str);

                While port_str.len > Ø do
                        [
                        port_wrd := Gethex (port_str);

                        data := Portin (port_wrd);
                        Puthex (data, hex_str);
                        Writeln (hex_str);

                        Readln (port_str);
                        ];
                ];

{-------------------------------------------------------------------}
PORTOUT mode   Get the port number and the data from the user.
{-------------------------------------------------------------------}

        If mode = 'out' then
                [
                Writeln ('Enter port number in hex');
                Readln (port_str);

                While port_str.len > Ø do
                        [
                        port_wrd := Gethex (port_str);
                        Write ('Data ');
                        Readln (hex_str);
                        data := Gethex (hex_str);
                        Portout (port_wrd, data);

                        Writeln ('Enter port number in hex');
                        Readln (port_str);
                        ];
                ];

        Write ('Continue (y/n) ');      Readln (mode);
    Until mode = 'n';
End.
```

Figure 5-15 PINPOUT.PAS Port I/O Demonstration Program.

Assembly Language IN and OUT

Now that we already have a demonstration program for PORTIN and PORTOUT, we'll take a brief look at the assembly language routines. A very brief look in fact, because this book is about Pascal, not assembly language.

Figures 5-16 and 5-17 show the assembly language listing or source files for the two routines PORTIN and PORTOUT. In order to use them in a Pascal program, the routines must be assembled into object files which can then be linked into our Pascal program.

The source files for assembly language programs are created in much the same way as the source files for Pascal programs: they are typed in with a word processing program. However, instead of the extension PAS given to Pascal source files, assembly language files must be given the extension ASM. Thus the first step is to create the source files PORTIN.ASM and PORTOUT.ASM from the listings in Figures 5-16 and 5-17.

Figure 5-16. Port IN instruction for Pascal

```
;****************************************************************
;PORTIN.ASM     Performs machine IN instruction.
;----------------------------------------------------------------
;This routine can be called from a Pascal program to
;perform the machine language IN instruction.
;----------------------------------------------------------------
coder   segment byte public

assume  cs:coder

public  portin

portin  proc    far

        push bp
        mov  bp,sp

        mov  dx,[bp + 6]

        in   al,dx

        pop  bp
        ret  2
```

```
portin  endp
coder   ends
        end

;-----------------------------------------------------------------
;Figure 5-16    PORTIN.ASM      Assembly IN Instruction.
;*****************************************************************
```

Figure 5-17. Port OUT instruction for Pascal

```
;*****************************************************************
;PORTOUT.ASM    Routine performs machine OUT instruction.
;-----------------------------------------------------------------
coder   segment byte public

assume  cs:coder

public  portout

portout proc far

        push bp
        mov  bp,sp

        mov  dx,[bp + 8]
        mov  ax,[bp + 6]

        out  dx,al

        pop  bp
        ret  4

portout endp
coder   ends
        end

;-----------------------------------------------------------------
;Figure 5-17    PORTOUT.ASM     Assembly Language OUT Instruction.
;*****************************************************************
```

The next step is to *assemble* these files to creat object files. This process is similar to using the Pascal compiler, except that it is a one-step process. Of course, to assemble an assembly language program you will need an additional program, the IBM Macro Assembler, a separate programming package (probably available from the same place you bought your Pascal compiler).

Figure 15-17 shows the prompts for assembling the routines PORTIN and PORTOUT.

Figure 5-18. Assembling the port routines

```
A>asm portin
The IBM Personal Computer Assembler
Version 1.ØØ (C)Copyright IBM Corp 1981

Object filename [PORTIN.OBJ]: Enter
Source listing  [NUL.LST]: Enter
Cross reference [NUL.CRF]: Enter

Warning Severe
Errors  Errors
Ø       Ø

A>asm portout
The IBM Personal Computer Assembler
Version 1.ØØ (C)Copyright IBM Corp 1981

Object filename [PORTOUT.OBJ]: Enter
Source listing  [NUL.LST]: Enter
Cross reference [NUL.CRF]: Enter

Warning Severe
Errors  Errors
Ø       Ø
```

The next step is to use the LINKer program to link the assembled routines and the compiled Pascal program (all OBJ files), into one executable program, the EXE file. Figure 5-19 illustrates how the LINKer is run. Notice that all three of the object file names are specified to the LINKer prompt for OBJ files.

The result of all this effort is the complete PINPOUT program, which can be executed directly from DOS like any other EXE file.

Figure 5-19. Linking the port routines

```
A>b:link

IBM Personal Computer Linker
Version 1.1Ø (C)Copyright IBM Corp 1982

Object Modules [.OBJ]: pinpout portin portout
Run File [PINPOUT.EXE]:          ← Just press Enter
List File [NUL.MAP]:             ← twice
Libraries [.LIB]: b:             ← Enter Drive containing PASCAL.LIB
```

Ports of Call

Here are some input/output ports that may be interesting to explore with our new PINPOUT demonstration program. The IN process will be harmless enough, but remember that the OUT instruction is a lot like a POKE. Things will be changed as a result of the OUT, so be prepared for possible problems should you inadvertently crash the system. It's best to have an *IBM Personal Computer Technical Reference* manual close at hand if you're going to experiment with these new routines.

Here are some input/output ports you may want to experiment with:

Hex Port Address	Description
40-43	8253-5 timer chip
60-63	8255A-5 timer chip
200-20F	game I/O adapter
378-37F	parallel printer port
380-38F	monochrome display CRT controller
3D0-3DF	color graphics adapter
3F8-3FF	RS-232-C ports

Tooting Your Own Horn

One interesting experiment you can try with PINPOUT is to make the speaker beep. There are two methods for generating sound with the speaker in the IBM PC. The speaker is controlled by a combination of

two timer chips. One of them is used to contain a *delay count* which will determine the audio frequency, or pitch, of the beep. The other timer is used to *gate* the speaker on and off. Both of these timers are accessible through the use of PORTIN and PORTOUT. Here is the basic idea.

Hex Port Address	Description
42	Through this port is entered the count which will determine the frequency of the beep. The count is a 16-bit value that is entered with 2 OUT instructions (least significant byte first). The larger the value, the lower the pitch.
61	This port is multi-purpose. Some of the 8 bits are used for the speaker while others are not.
	Bit 0 = 1 Use timer (port 42) for count. = 0 Pulse speaker from bit 1.
	Bit 1 = 1 Gate the speaker on. = 0 Gate the speaker off.
	Bit 2-7 Not used for speaker, but must be preserved when bit 0 or 1 is altered.

Bit 1 actually turns the speaker on and off. If bit 0 is on, the value found in the timer port will control the pitch. If bit 0 is off when bit 1 is changed, only one pulse to the speaker will occur. This means that you can create your own pitches by pulsing bit 1 on and off, while keeping bit 0 off.

The easiest way to beep the speaker is to first program a delay count into the timer through port 42, and then toggle the speaker, that is, turn it on through port 61. We want to make sure to preserve the other 6 bits of port 61, so the first thing we should do is use the IN command to establish their value. Here's how to run the program.

```
A>pinpout
Enter in or out ---> in
Enter port number in hex ---> 61
ØØ4d
```

So that is how we have to restore port 61 after we're done tooting around. Now, let's use the program to make some noise.

```
Continue (y/n) ---> y
Enter in or out ---> out
Enter port number in hex ---> 42
Data ---> ØØ
Enter port number in hex ---> 42
Data ---> 2Ø
```

There, that much will put a delay count of 2000h into the timer. This is 8,192 in decimal, and represents about 1/8th of the timer range. Next, we need to toggle the speaker.

```
Enter port number in hex ---> 61
Data ---> 4f
```

There's your beep! Since the value at port 61 was equal to 4d, to turn on bit 1 we must OUT a value of 4f. This is because the binary value of 4d is 01001101, in which bit 1 (the second from the right) is set to zero. Turning this bit on changes the binary value to 01001111, which is 4f hexadecimal.

Changing this bit to "on" tells the speaker chip to turn on the speaker and use the timer count to determine the pitch. Let's try changing the pitch while it's running.

```
Enter port number in hex ---> 42
Data ---> ØØ
Enter port number in hex ---> 42
Data ---> 7
```

A higher pitch. We have entered a smaller delay count into the timer. This means that the time it takes to count down is shorter; therefore the frequency is higher. Let's try one more.

```
Enter port number in hex ---> 42
Data ---> ØØ
Enter port number in hex ---> 42
Data ---> 3
```

An even higher pitch. We've just about halved the timer count. Would you say the pitch is twice as high? Getting tired of listening to it? Okay. Let's turn it off.

```
Enter port number in hex ---> 61
Data ---> 4d
```

Ah... silence is golden.

Blaise Away!

So now you have the tools to make sound. Things can be automated of course, so we don't have to sit here typing in hexadecimal values. We can do it all from our program. Shown in Figure 5-20 is a simple example program which produces a sound effect using Pascal. We call this program BLAISER BLAST.

Figure 5-20. Blaiser Blast sound demonstration program

```
{*****************************************************************}
BLAISER.PAS     Blaiser Blast sound demo
{-----------------------------------------------------------------}
       This program uses the PORTIN and PORTOUT routines to
operate the sound timer circuits in the IBM PC.
{-----------------------------------------------------------------}

Program Blaiser (input,output);

Const
    speaker     = #61;          { speaker toggle port address
    timer       = #42;          { timer port address
    toggle_on   = #4f;          { value to turn speaker on
    toggle_off  = #4d;          { value to turn speaker off
```

```
    max_count    = 25Ø;
    scaler       = 2;

Var
    code,
    count        :word;
{
-------------------------------------------------------------------
       Include the assembly language routines as external
-------------------------------------------------------------------
}
{$include 'portio.pas'}

-------------------------------------------------------------------
       Main program.  Another endless loop.  Every time the ENTER
key is pressed, the "blaiser" fires away.  Try just holding down
the ENTER key and see what happens.
-------------------------------------------------------------------
}
Begin
  Repeat
    Write ('<RETURN> to fire');
    Readln;

    Portout (speaker, toggle_on);  { toggle the speaker on      }

    For count := Ø to max_count do { change the delay           }
        {
        code := Sqr (count) Div scaler;
        Portout (timer, lobyte (code) );
        Portout (timer, hibyte (code) );
        };

    Portout (speaker, toggle_off); { toggle the speaker off     }

  Until count = #ffff;
End.

-------------------------------------------------------------------
Figure 5-2Ø     BLAISER.PAS     Blaiser Blast Sound Demo.
*******************************************************************
```

Since a laser blast sound should have kind of a sweeping quality, we will generate the sound in a FOR structure which will be converted into an exponential generator through the use of the SQR function. We'll just throw in some constants to get started. You might want to fool around with changing these to see what happens to the sound.

Notice how the speaker is toggled on, outside of the FOR structure. All that happens within the structure is the rapid change of frequencies

to be ported out to the timer one byte at a time. Finally, the speaker is toggled off after the program exits from the FOR structure.

The whole thing is suspended inside a REPEAT structure that will never be true. This will allow the program to continue until you abort it using Ctrl C.

Well, it's up to your imagination now. Sound is available to your Pascal programs using a small external assembly language routine to provide IN and OUT instructions. These routines PORTIN and PORTOUT can be very useful in the graphics area as well.

Access to the CRT Controller

Remember earlier in this chapter we did some simple monochrome graphics manipulation and we couldn't get past the CRT controller's ported access? PORTIN and PORTOUT should solve that problem. We now have the tools to IN or OUT to any port address. Let's explore what we can do with the CRT controller now.

CRT Controller Ports and Registers

The 6845 controller chip used in the IBM-PC has several ports connected to it. It is programmable and all of the programming happens through two ports. One, the *index port*, is used to specify one of the 18 registers inside the chip. We can access only 1 of the 8-bit registers at a

3B0	Not Used
3B1	Not Used
3B2	Not Used
3B3	Not Used
3B4	6845 Index Register
3B5	6845 Data Register
3B6	Not Used
3B7	Not Used
3B8	CRT Control Port 1
3B9	Reserved
3BA	CRT Status Port
3BB	Reserved
3BC	Parallel Data Port
3BD	Printer Status Port
3BE	Printer Control Port
3BF	Not Used

Table 5-1. Port addresses for CRT controller

time through the other port, the *data port*. The port addresses for the CRT controller are shown in Table 5-1.

To access one of the 6845's registers, we must first OUT the register number to the index port 3B4. Then, we may OUT the 8-bit value we wish to load to that register, through the data port 3B5. There are several things about the CRT that we can control with this method. Notice that a lot of the registers shown in Table 5-2 are "write only." That

REG. ADDR	DESCRIPTION	UNITS	I/O	40x25 ALPHA	80x25 ALPHA	GRAPHIC MODES
0	Horizontal Total	Char.	Write Only	38	71	38
1	Horizontal Displayed	Char.	Write Only	28	50	28
2	Horiz. Sync Position	Char.	Write Only	2D	5A	2D
3	Horiz. Sync Width	Char.	Write Only	0A	0A	0A
4	Vertical Total	Char. Row	Write Only	1F	1F	7F
5	Vertical Total Adjust	Scan Line	Write Only	06	06	06
6	Vertical Displayed	Char. Row	Write Only	19	19	64
7	Vert. Sync Position	Char. Row	Write Only	1C	1C	70
8	Interlace Mode	—	Write Only	02	02	02
9	Max Scan Line Addr.	Scan Line	Write Only	07	07	01
A	Cursor Start	Scan Line	Write Only	06	06	06
B	Cursor End	Scan Line	Write Only	07	07	07
C	Start Addr. (H)	—	Write Only	00	00	00
D	Start Addr. (L)	—	Write Only	00	00	00
E	Cursor Addr. (H)	—	Read/ Write	XX	XX	XX
F	Cursor Addr. (L)	—	Read/ Write	XX	XX	XX
10	Light Pen (H)	—	Read Only	XX	XX	XX
11	Light Pen (L)	—	Read Only	XX	XX	XX

Note: All register values are given in hexidecimal.

Table 5-2. 6845 registers

means that values are OUTed to them, but nothing can be INed. Some of them have to do with the number of lines and columns on the screen. Others have to do with the cursor.

Positioning the Cursor. If we want to control the cursor position (and make our Pascal programs appear "smarter") we need to look at the *cursor address* registers. There are two of them; one to hold the most significant byte, and the other to hold the least significant byte of the cursor address. The cursor address can be thought of as the character position on the screen where the cursor presently exists. Start in the upper left hand corner at column 0 on line 0, and call that position 0. Moving to the right on line 0, keep numbering up to 79, (assuming you are using the 80-column monochrome display). Now, just as if you were reading, you drop to the leftmost end of the next line and call that position 80, and so on.

As we've said, there are 25 lines of 80 columns on the monochrome display, adding up to a total of 2,000 character positions. A number that large requires 16 bits to store, and therefore requires two registers in the 6845 CRT controller. To get a quick idea of how to position the cursor, take a look at the procedure shown in Figure 5-21.

Figure 5-21. Cursor positioning procedure

```
{*************************************************************
CURSPOS.PAS     Monochrome cursor positioning procedure
-------------------------------------------------------------
        This procedure will position the cursor of the monochrome
display to the column and line coordinates passed as parameters.
-------------------------------------------------------------
}
Procedure Curspos (Const col, line:word);

Const
        count           = 80;
        index_reg       = #3B4;
        data_reg        = #3B5;
        curs_h_reg      = #0E;
        curs_l_reg      = #0F;

Var
        curs_wrd        :word;

Begin
        curs_wrd := count * line + col;

        Portout (index_reg, curs_h_reg);
        Portout (data_reg, Hibyte (curs_wrd) );
```

```
        Portout (index_reg, curs_l_reg);
        Portout (data_reg, Lobyte (curs_wrd) );
End.

{-------------------------------------------------------------
Figure 5-21     CURSPOS.PAS     Cursor Positioning Procedure.
*************************************************************}
```

There are two parameters to this procedure: the column position and line number where we want the cursor to be. In a CONST section we have set up the port numbers for the index and data registers of the 6845. Also, we have set up the register numbers for the high and low bytes of the cursor address.

To calculate the relative location from the column and line coordinates, we do some simple arithmetic. This gives us a 16-bit relative cursor position in the variable "curs_wrd." It takes four OUT instructions to position the cursor. The first two set up the index to point to the high byte register and then OUT the data using the HIBYTE function. Then, in a similar manner, the low byte of the address is OUTed. That's it. The cursor is now at the new position, as you can see by running the program set forth in Figure 5-22. It is a demonstration program that will pass the cursor from upper left to lower right on the screen.

Figure 5-22. Cursor positioning demonstration program

```
{*************************************************************
POSTEST.PAS     Cursor positioning demo
-------------------------------------------------------------
        This program demostrates the use of the procedure
CURSPOS.  Just to keep things simple, we will only flash the
cursor around the screen.
-------------------------------------------------------------}

Program postest;

Const
        delay_cnt       = 1000;

Var
        delay,
        col,
        line            :word;
```

```
{-------------------------------------------------------------------
    Include the port routines as well as the cursor routine
-------------------------------------------------------------------}

{$include:'portio.pas'}
{$include:'curspos.pas'}

{-------------------------------------------------------------------
        Main program.  Place the cursor in every position on the
screen.
-------------------------------------------------------------------}
}
Begin

        For line := Ø to 24 do      { switch order of FOR statements }
            For col := Ø to 79 do   { to perform vertical sweep }
                {
                Curspos (col, line);
                For delay := 1 to delay_cnt do
                    [];
                };
End.

{-------------------------------------------------------------------
Figure 5-22     POSTEST.PAS     Cursor Position Demo Program.
*******************************************************************}
```

Observe the "$include" metacommands again. These are hooking up some of these procedures and functions for us. It's a simple enough program. Just two nested FOR structures, one representing columns, the other lines. Together, they sweep the cursor across each line, and on down the screen.

Locating the Cursor. Occasionally it is useful to know where the cursor is on the screen. Notice that in the table of registers for the 6845, the cursor address registers are read/write type. This means that you can use the PORTIN function to get the value of the cursor position from the controller. To accomplish this we will use another procedure to get the cursor location for us, and present it in column and line form.

The routine shown in Figure 5-23 accesses the two registers on the 6845 controller that contain the cursor address. Each byte of the cursor address is obtained by combining the use of PORTIN and PORTOUT. Then, the two bytes are passed to the IBM function BYWORD to convert them into a WORD type. From this, the column and line can be calculated using the simple arithmetic functions DIV and MOD. Now, we need a program to test this CURSLOC procedure.

Figure 5-23. Cursor location procedure

```
{*****************************************************************}
CURSLOC.PAS     Monochrome cursor locate procedure
{-----------------------------------------------------------------}
        This procedure determines the position of the cursor on
the monochrome display.  It uses the PORT routines to get the
position from the CRT controller, and returns it in the form of
column and line.
{-----------------------------------------------------------------}
;
Procedure Cursloc (Var col, line:word);

Const
        count           = 80;
        index_reg       = #3B4;
        data_reg        = #3B5;
        curs_h_reg      = #0E;
        curs_l_reg      = #0F;

Var
        curs_h,
        curs_l          :byte;

        curs_wrd        :word;
!
{-----------------------------------------------------------------}
   Main program.  Get the high and low bytes of the position.
{-----------------------------------------------------------------}
!
Begin
        Portout (index_reg, curs_h_reg);
        curs_h := Portin (data_reg);

        Portout (index_reg, curs_l_reg);
        curs_l := Portin (data_reg);

!
{-----------------------------------------------------------------}
              Now, calculate the line and column.
{-----------------------------------------------------------------}

        curs_wrd := byword (curs_h, curs_l);
        line := curs_wrd Div count;
        col  := curs_wrd Mod count;

End;
!
{-----------------------------------------------------------------}
Figure 5-23     CURSLOC.PAS     Cursor Location Procedure.
{*****************************************************************}
```

The program shown in Figure 5-24 uses the standard Pascal routines READLN and WRITELN to perform input and output. The idea is to display a specified number of asterisks by entering the count. Each asterisk is output one at a time using the WRITE procedure. This does not put a carriage return and line feed at the end, so the cursor will be left positioned at the end of the line of asterisks. The call to the CURSLOC procedure will then return the column and line position of the cursor. The process can be repeated as long as the count is greater than zero.

Figure 5-24. Cursor location demonstration program

```
{*****************************************************************
LOCTEST.PAS     Cursor location demo
-----------------------------------------------------------------
        This program demonstrates the use of the cursor location
procedure CURSLOC.PAS.  The user inputs a character count.  The
program then outputs that number of characters, and then calls
the procedure to determine the cursor position.  This should be
in agreement with the number of characters that were requested.
-----------------------------------------------------------------

Program Loctest (input,output);

Var
        col, line,
        pos, count      :word;
'
-----------------------------------------------------------------
    Include the PORT routines, and the cursor locate routine.
-----------------------------------------------------------------

{$include:'portio.pas'}
{$include:'cursloc.pas'}
'
-----------------------------------------------------------------
            Main program.  Get the character count.
-----------------------------------------------------------------

Begin
        Writeln ('Enter character count in decimal');
        Readln (count);

        While count > Ø                 { output the count      }
            Do  [
                For pos := 1 to count do
                        Write ('*');
```

```
            Cursloc (col, line);
            Writeln (col, line);

            Readln (count);
        end;

End.
```

Figure 5-24 LOCTEST.PAS Cursor Location Demo

Summary

Well...in this chapter we've added quite a bit of power to the Pascal language. We have filled in some of the holes in the language with functions and procedures to allow access to the monochrome display buffer and the CRT controller. We have also given Pascal a musical voice to cheer up our programs. This is only the beginning. With these tools you can go on to fully explore your IBM PC and Pascal.

Exercises

1. What is the difference between a function and a procedure?

2. What is an address variable?

3. What is the difference between value parameters and reference parameters?

4. What is an EXTERNAL function or procedure?

Solutions

1. A procedure is called as a statement by itself; a function is used in an expression or assignment statement.

2. An address variable is a 16-bit or 32-bit address of another type of variable.

3. Value parameters are placed on the stack; reference parameters have the address of a variable placed on the stack.

4. An EXTERNAL function or procedure must have its name and parameters declared while the rest of the function or procedure is in another file.

6

Input and Output with Files

Concepts

Files as structured data types
Components
Buffer variable
Accessing a file
Sequential and direct modes
End-of-file considerations
Terminal mode
Binary and ASCII files
Keystroke processing
General ledger and customer file systems
Loading, printing, and updating a file

Keywords

FILE, INPUT, OUTPUT, TEXT, READFN, ASSIGN, RESET, REWRITE, CLOSE, GET, PUT, READ, READLN, WRITE, WRITELN, SEEK, TERMINAL, DIRECT, SEQUENTIAL, EOF, .MODE

We've seen that Pascal is very thorough about dealing with data elements. If you spend a little time thinking out your program design, you can come up with a well-structured, efficient system using Pascal. The program will always know just what kind of information it is dealing with, and what operations are permitted. The only problem is that this data processing ability is relevant only to the particular program at hand. The data elements are *embedded* within the program (except for data entered at the keyboard or printed on the screen, which can never be a great deal of data), and can only be used by that program. Also, the data only exist as long as the particular program is running. When some other program is loaded, or the system power is turned off, the data elements are lost.

What Is a File?

What our Pascal programs need is some way to communicate with the world outside of the program, and to store information for further use. This need is common to all programs, regardless of their language. In Pascal, this need is fulfilled through the use of *files*. A file has two purposes.

INPUT and OUTPUT (I/O) to provide a communication channel to the outside world

STORAGE to save data for future use

Throughout this book, we have been using files for input and output. Remember the names "input" and "output" that appear in the PROGRAM declaration statement at the beginning of almost every program example we've used? These are special *predeclared files* which we will explore further. Generally, "input" is assigned to the keyboard and "output" to the display. Together, they provide interactive "conversation" with the user.

FILE Variable Declaration

The files "input" and "output" are declared automatically by using their names as parameters in the PROGRAM declaration. Other files may also be declared in a Pascal program. User-defined files are declared as variables in a VAR statement. That's because in Pascal, a FILE is just another structured data type. Here are some examples of how files might be declared.

```
Var
    numbers        :file of integer;
    prem_rates     :file of real;
    letter         :file of char;
```

Files can be thought of as similar to arrays, especially when considering files of simple types as above. Most of the time we will want our files on the disk, where they can be stored permanently.

More complex files can be declared through the combined use of the TYPE and VAR statements. Here is an example of how to declare a transaction file.

```
Type
     transaction = record
          tran_date      :string(8);
          tran_type      :char;
          tran_desc      :string(2Ø);
          tran_amount     real.
               end;

Var
     tran_file      :file of transaction;
```

File Components

The TYPE clause that is included in the FILE declaration above specifies what components constitute the file. In our earlier examples, each component had been a single ordinal element. In this more complex example, we're using a RECORD structure for each file component.

Each file component of "tran_file" is described in the TYPE declaration for "transaction." It is actually a RECORD consisting of several fields including the date (MM/DD/YY); a single character transaction type; a 20-character description; and the amount, carried as a REAL. As shown in Figure 6-1, this record occupies 34 bytes of memory.

Each time a program references a component of this file, it will really be referencing one entire record with all its fields. This is very important, since the program will only be able to access one component of a file at any given instant.

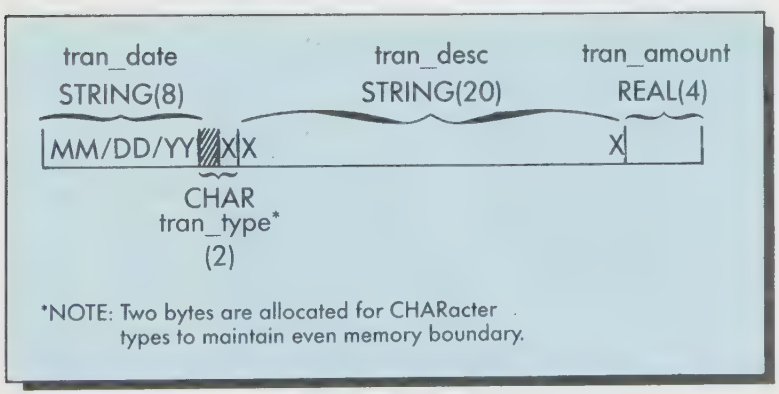


Figure 6-1. Transaction file record component

File Buffer Variable

All files in Pascal are accessed through the use of the *buffer variable*. This can be thought of as a "window" into the file that allows us to see one component at a time. (See Figure 6-2.)
The buffer variable is identified by the name of the file variable, followed by a caret (^). It may be used like a pointer variable in the sense that it can be assigned and used in expressions.

The IBM Pascal compiler doesn't like you to use a buffer variable in certain situations. In general, it is better programming practice to routinely declare a second variable with the same type as the file component. Then, before processing a component of the file, you can assign the component to the variable. Do all your processing on this variable, and then assign its value back to the buffer variable just before outputting it to the file.

```
Var
     tran_input      :transaction;
     tran_file       :file of transaction;

Begin
          { Input the component from the file }
     tran_input := tran_file^;
     With tran_input Do
          { Process fields in variable component }
     tran_file^ := tran_input;
          { Output the component to the file }
End.
```

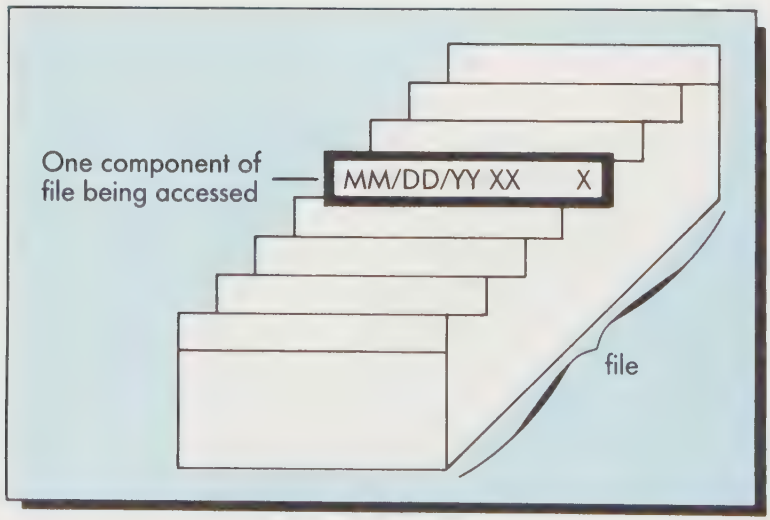


Figure 6-2. Buffer variable is like a window

This approach will ensure that data are not lost as a result of accidentally reassigning the buffer variable before output is performed. It will also prevent some compiler warnings that are issued when the buffer variable is used.

File Types

In reality, all file components, all data elements, all programs and operating systems consist of bits — nothing but 1s and 0s arranged in some meaningful order. The way we interpret the bit arrangements lends meaning to their patterns. In IBM Pascal, there are two ways of looking at the bits that compose a file structure.

BINARY	All data types can be represented in binary, just like in memory.
TEXT or ASCII	Data are translated to or from ASCII characters.

Binary Files

The components in a BINARY file are represented exactly the same way as they are in memory. For example, an INTEGER component would occupy two adjacent 8-bit bytes, a REAL component would occupy four bytes, and a STRING(n) would occupy (n) bytes. A RECORD type can also be a component of a binary file. The individual fields are connected together and can be input or output as simply a group of bytes.

A binary file is also called unformatted since there is no character conversion involved in the transfer. (See Figure 6-3 for a representation of a binary or unformatted file.) This type applies mostly to files which reside either on disk or tape. The component size in bytes is determined by the total of all the fields.

TEXT or ASCII Files

TEXT is a predeclared file type in Pascal. It is a special file which is comprised of ASCII characters. (See Figure 6-4.) Files of this type are compatible with word processors, or other character-oriented programs like compilers. These files are *line oriented*. The components are actually ASCII characters which are separated into *lines*. Lines are determined by a special character known as the *line marker*, which corresponds to a carriage return and line feed on most I/O devices.

Since a TEXT file is comprised only of ASCII characters, we must do some conversion to allow numeric types to be components. Outputting

the 16-bit representation for an INTEGER type to a TEXT file would result in 2 characters of unknown value. The INTEGER must first be converted into a string of characters that represent its value. This is done automatically by the procedures READ, READLN, WRITE, and WRITELN which are used to process TEXT files.

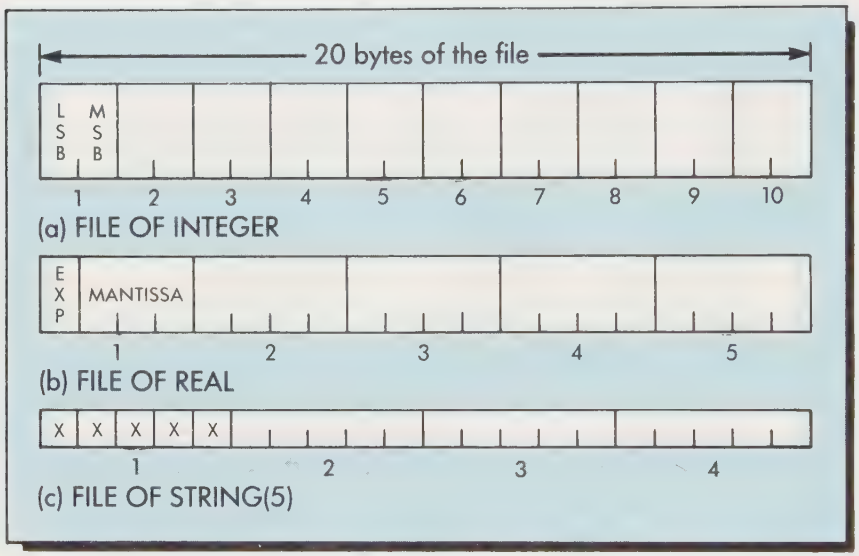


Figure 6-3. Binary or unformatted files

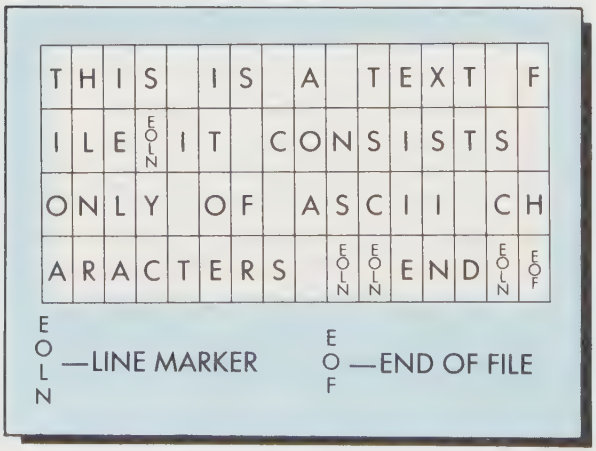


Figure 6-4. Text or ASCII or formatted files

File Access Routines

Pascal provides several different methods of utilizing files. File handling is performed through the use of predeclared functions and procedures, each applicable in different situations. These functions and procedures use the DOS routines to perform the I/O.

We'll be discussing the use of each of these routines in this chapter. For now, we'll just give you a quick idea of how many routines there are and what they do.

Initializing and Terminating Files

You will need certain routines to initialize and terminate files.

READFN	Reads DOS file name and assigns to Pascal file.
ASSIGN	Assigns name in STRING variable to Pascal file.
RESET	Opens a Pascal file for input.
REWRITE	Opens a Pascal file for output.
CLOSE	Closes a Pascal file.

Unformatted Input and Output

There are two routines you will use for unformatted I/O.

GET	Inputs one component with no conversion.
PUT	Outputs one component with no conversion.

Formatted Input and Output

There are four routines you will use for formatted I/O.

READ	Inputs characters and assigns a variable.
READLN	Like READ, but it ignores balance of line.
WRITE	Encodes a variable and outputs characters.
WRITELN	Like WRITE, but it includes a carriage return.

Direct Access

There is one routine you will use for direct access.

SEEK	Prepares for I/O with a specified component.

File Modes

We use the window provided by the buffer variable to input or output one component of a file at a time. The manner in which the one component is selected is determined by the *mode* of the file. There are three file modes used by Pascal.

TERMINAL	Components are read or written in real time usually from the keyboard and to the display.
SEQUENTIAL	Components are accessed in a series beginning with the first and continuing to the last, with no backing up or skipping around.
DIRECT	Components are accessed in any order using the SEEK procedure.

Terminal Mode

So far in this book we have been using a special type of file called a *TERMINAL* mode file. A TERMINAL mode file is one which is interactive with the user, generally using the keyboard and display devices. A TERMINAL mode file is illustrated in Figure 6-5. In Pascal,

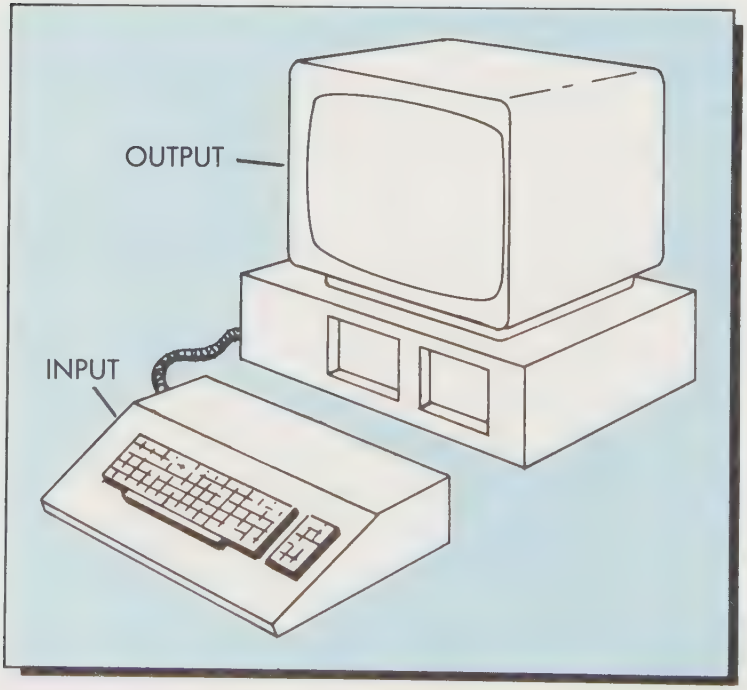


Figure 6-5. Terminal mode files

the predeclared files "input" and "output" are assigned to the keyboard and display respectively. TERMINAL mode files are similar to SEQUENTIAL files and must be of ASCII type. This means that the file components are characters input or output one at a time.

Sequential Mode

When a file is in SEQUENTIAL mode the components are accessible only in the order that the file was created. Accessing begins with the first component in the file and continues from one component to the next. There is no ability to back up (except all the way to the beginning), and random accessing of the components is not possible.

End-of-File in Sequential Mode

At the end of a sequential file there is a special component called the EOF (end-of-file). It provides a means of detecting the end of a file in a Pascal program. The EOF component is placed in the file when it is created, and can be used from then on to signal the end-of-file condition. This condition is available to the Pascal program in the form of a Boolean function called EOF, which is returned TRUE if the end of file has been reached. This allows a nice form for sequential processing.

```
While not Eof (file_name) Do...
```

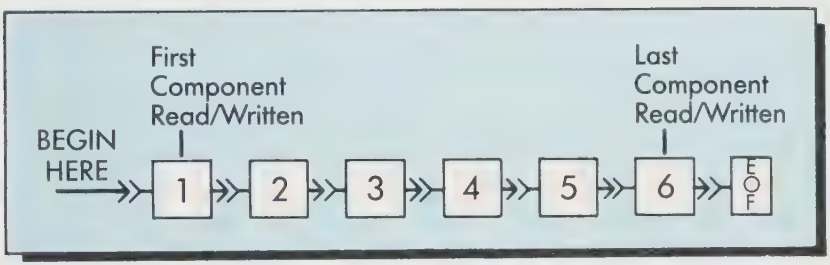


Figure 6-6. Sequential access

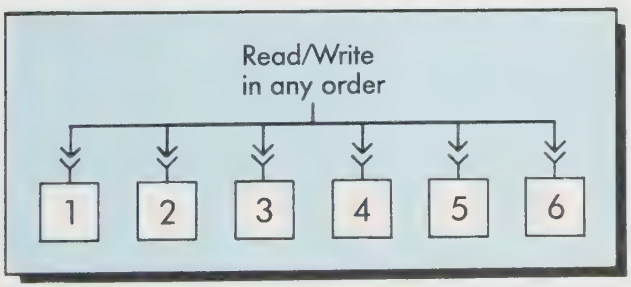


Figure 6-7. Direct access

Direct Mode

DIRECT mode access to a file permits the program to operate on *any* component in the file, in random order. (See Figure 6-7.) The *target* component is specified by its *component number* in the file. This requires that the program "know" where the desired component exists within the file. It also implies that the file resides on some hardware medium that is physically capable of random access. For example, it would not make sense to declare the printer as a direct mode file. Since the printer's output must be one component (character) at a time, it can only be used as a sequential file.

Usually, direct mode files will be on disk. The components will be stored as a group of bytes in IBM disk format. The DOS routines used to access the file will know how many bytes are in each component and where each component is located on the disk.

To access a file in direct mode, you must set the MODE to "direct". This sets a flag in the *file control block* that determines which DOS routines will be used to access the disk file. This must be done prior to any other operation with the file, using the assignment statement:

```
file_name.mode := direct;
```

End-of-File in Direct Mode

The *IBM Personal Computer Pascal Compiler* manual warns us what happens to the EOF condition when a file is accessed in direct mode. According to the manual, when a file is created (or updated) in DIRECT mode, the end-of-file is not exact. Thus the normal EOF can't be used to detect the true end-of-file. Not only that, but the file's EOF will no longer be detectable in SEQUENTIAL mode either once it has been created or updated in DIRECT mode.

Why does this matter? Once a disk file has been accessed using DIRECT mode, it can no longer be accessed using SEQUENTIAL mode. This is rather inconvenient to say the least, and we hope some improvement is forthcoming in a revised Pascal compiler for the IBM PC.

It seems to us that the best solution for now (until IBM comes out with a revised compiler) is to always access your file in one mode or the other. This may seem like extra work, especially just for printing listings or other normally sequential processes; but if you ever use DIRECT mode on a file, its EOF will be undependable for SEQUENTIAL mode access. Better some extra work today than an unuseable file tomorrow!

Terminal Mode Files

We have been using TERMINAL mode files in our examples all through this book without fully explaining their operation. Every time we use READLN to input some data for a test program, we are actually using the system keyboard as an *input file*. The procedure reads characters from the keyboard and converts them into the appropriate variable types for the parameters in the READLN calling sequence. Correspondingly, every time a program produces a prompt on the display using WRITELN, it is actually using the monochrome display unit as an *output file*.

Predeclared Files: INPUT and OUTPUT

The file names "input" and "output" appear in the PROGRAM declaration of most of our program examples. In Pascal, these are predeclared files that are automatically assigned to the system console. Since all Pascal I/O utilize the DOS routines, these two files operate in SEQUENTIAL mode, transferring one character at a time.

As characters are typed on the keyboard, they are echoed automatically to the display so that you can see what you are typing. Characters are buffered by the DOS so that nothing is actually sent to the program until you press the Enter key. This permits line editing with the Bksp and Del keys and provides some degree of input validation, since entering even one character will require two keystrokes (the key itself and Enter).

Keystroke Processing

In some cases, it would be nice to be able to make things happen in our Pascal programs by pressing a single key, instead of having to always press Enter as well. For example, suppose we want our program to display a menu of operations from which we are to select. We would like the selection to be executed by pressing a single key. The menu might look like this:

```
***** Main System Menu *****

File Maintenance .....   f
Order Entry ..........   o
Sales Analysis .......   s
Utility Programs .....   u

Enter Selection -------->
```

Just to keep this example simple, when one of the menu selections is entered, we'll redisplay the option name to indicate positive selection. If an invalid key is pressed, the menu will simply be re-displayed.

In the example provided in Figure 6-8, we use several of the file handling routines to give you an idea of how they work. It's important to notice how the file for the keyboard is handled. In the VAR section notice the declaration for the file "kybd" as a FILE OF CHAR. Right after the BEGIN statement for the main program, we use the ASSIGN procedure to assign the name "user" to the Pascal file "kybd". USER is a special file name used for nonbuffered keyboard input on the IBM PC. Individual keystrokes are available from this input file.

Figure 6-8. Interactive menu demonstration

```
{
*****************************************************************
MENU.PAS        Interactive Menu Demonstration
-----------------------------------------------------------------
        This program is intended to illustrate the use of
keystroke-oriented responses to menus in general.  The input
is performed by the user procedure "Readkey."
-----------------------------------------------------------------
}
Program Menu (input,output);

Const
  item_cnt = 4;

Type
  text_line = lstring(30);
  index     = 1..item_cnt;

Var
  heading       :array[1..2] of text_line;
  option        :array[1..item_cnt] of text_line;
  code          :array[1..item_cnt] of char;
  select        :text_line;
  answer        :char;
  item          :index;

  kybd          :file of char;  { <----- File declaration
```

```
{-------------------------------------------------------------------
        This section contains the information on the menu
which is to be displayed.  There are better ways to handle
menus, such as storing them in disk files, but this will do
for now.
-------------------------------------------------------------------}
Value
        heading[1] := '***** Main System Menu *****';
        heading[2] := 'Subsystem ............ Enter';

        option[1]  := 'File Maintenance .....    '   ;
        code[1]    :=                           'f' ;
        option[2]  := 'Order Entry ..........    '   ;
        code[2]    :=                           'o' ;
        option[3]  := 'Sales Analysis .......    '   ;
        code[3]    :=                           's' ;
        option[4]  := 'Utility Programs .....    '   ;
        code[4]    :=                           'u' ;

        select     := 'Enter Selection -------> '   ;

{-------------------------------------------------------------------
        This function uses the file routine GET to access the
CHAR type components of the keyboard file (kybd).  The function
loops until an ASCII character appears in the buffer.
-------------------------------------------------------------------}
Function Readkey        :char;
    Begin
        Repeat Get (kybd) Until kybd^ <> chr(Ø);
        Readkey := kybd^;
    end;

{-------------------------------------------------------------------
        This is the main program.  There is no way out.
Assign the "kybd" file to the non-buffered keyboard routine
in the BIOS.
-------------------------------------------------------------------}
Begin
        Assign (kybd,'user');    { <---- Non-buffered keyboard   }
        Reset (kybd);
```

```
{-------------------------------------------------------------------}
        This section displays the entire menu each time
through the program, and prompts the user for a selection.
{-------------------------------------------------------------------}

    Repeat
        Writeln (heading[1]);
        Writeln (heading[2]);
        Writeln;
        Writeln (option[1], code[1]);
        Writeln (option[2], code[2]);
        Writeln (option[3], code[3]);
        Writeln (option[4], code[4]);
        Writeln;
        Write   (select);

        answer := Readkey;      { <---- Keystroke input         }

{-------------------------------------------------------------------}
        If the selection is valid, re-display it for
verification, otherwise just loop back.
{-------------------------------------------------------------------}

        For item := 1 to item_cnt Do
                If answer = code[item]
                    Then Writeln (option[item]);

{-------------------------------------------------------------------}
                            No way out.
{-------------------------------------------------------------------}

    Until false;
End.

{-------------------------------------------------------------------}
Figure 6-8.     MENU.PAS  Interactive Menu Demonstration Program.
*******************************************************************
```

The trick is that we have to be able to tell when a key is being pressed. This is done by continuously inputting from the file "kybd" until a value other than CHR(0) appears in the buffer variable. In our example, this is achieved by our user-written function "Readkey". But the real work is done when "Readkey" calls the file routine GET. That's just what it does: it gets a component from the file specified as a parameter. In this case, the file is "kybd", and one component is one character. "Readkey" hangs inside the REPEAT structure until a character appears. It is then assigned to the function identifier to be used by the program.

Immediately following the ASSIGN statement is a RESET statement. This initializes the file for input, and actually performs the first GET. This is a CHR(0) unless a key was being pressed just as the RESET was executed.

The rest of the program is unspectacular, just a quick way of displaying a menu. One thing you might see is that there is no way out of this one (except Ctrl Break). But then the program doesn't really do anything either.

SEQUENTIAL Mode Files

Most of the time, when we hear the word "file," we think of a disk file containing data to be used by some program. This is the best way to save information over a period of time. In standard Pascal, all files are *SEQUENTIAL*; each component must be considered one after another, in order. This concept dates back to the days when computers did everything with punched cards or magnetic tapes. Files were often kept in card form, sometimes occupying many drawers. The computing machinery couldn't just pluck one card out of the middle of the file for processing. Instead, all of the cards had to be processed to find a desired component in the middle of a file.

This principle worked for years in the data processing world, and it still works adequately for some applications. The sequential approach can be quite satisfactory when the number of components in a file is small. If the file is on a disk, it won't take long to process all of the records in sequence each time processing is required.

In the next two examples we'll demonstrate how to create a sequential file. We'll deal with both a BINARY file and a TEXT file and compare their relative merits.

General Ledger File

If you're interested in accounting using your IBM PC, this example is for you. The idea is to set up a file on the disk to represent a General Ledger. The file components will be RECORDs, each record containing the information for one General Ledger (G/L) account. The declaration for the fields within each record is done in a TYPE statement. We have created the TYPE declaration as a separate file which can be $INCLUDEd in any compilation.

Figure 6-9 shows a typical General Ledger account. There are several different types of data involved with this record. The account number is an INTEGER. There's a CHAR type code, two STRINGs and a whole lot

of REAL types to store dollar amounts. This type of record structure would provide the hub of a home accounting system.

Figure 6-9. General ledger master file record

```
{
*****************************************************************
GLMASTER.REC    G/L Master Field Declarations
-----------------------------------------------------------------

Type
  gl_master = record
    acct_no     :integer;        { Account number for reference  }
    acct_type   :char;           { 'a':asset, 'l':liability, etc.}
    acct_desc   :string(2Ø);     { Descriptive title             }
    last_activ  :string(8);      { Date MM/DD/YY last activity   }
    last_tran,                   { Last transaction amount       }
    beg_bal,                     { Beginning balance             }
    end_bal,                     { Ending balance                }
    curr_debit,                  { Total debits this period      }
    curr_credit                  { Total credits this period     }
                :real;
    hist_debits,                 { Total debits history array    }
    hist_credits                 { Total credits history array   }
                :array[1..hist_size] of real;
end; {gl_master.rec}
{
-----------------------------------------------------------------
Figure 6-9.     GLMASTER.REC  General Ledger Master File Record
*****************************************************************
```

Loading the G/L File

The next thing we need to do is write a program to load the G/L file with the accounts we will be using. In order to construct a file of records that can be used as a general ledger, we must have a program that *initializes* all of the fields in each record just the way we want them. We will want to assign our own account numbers and descriptions, and be able to set up beginning and ending balances. This program, set forth in Figure 6-10, is called "Glload".

First look at the way the field declarations for "gl_master" are "$INCLUDEd" in the compilation right in the beginning of the declarative section. Once the type "gl_master" has been declared, the variable "master" is declared, as well as the file "glmaster."

The DATE procedure is declared here as EXTERNAL: the way that IBM (for some reason) requires it. Assuming our system has been initialized with the correct date, each G/L account record we create will have today's date in it.

Figure 6-10. General ledger file loading program

```
{*************************************************************************}
GLLOAD.PAS      Load General Ledger File
-------------------------------------------------------------------------
        This program loads the General Ledger file.  The
data for each account are entered from the keyboard, while
the file is constructed in sequential mode.
-------------------------------------------------------------------------
}
Program Glload (input,output);

Const
  glname = 'glmaster.dat';       { <---- DOS file name is CONST  }
  end_input = Ø;
  hist_size = 12;

{$include:'glmaster.rec'}        { <---- Compiles TYPE declaration }

Var
  today          :string(8);
  acct_in,
  index          :integer;

  master         :gl_master;     { <---- RECORD type variable    }

  glmaster       :file of gl_master;     { <---- FILE declaration }


-------------------------------------------------------------------------
                Date procedure declared external
-------------------------------------------------------------------------

Procedure Date (Var s string : external.

-------------------------------------------------------------------------
        Main program begins here.  Get the date, and open
the General Ledger master file.
-------------------------------------------------------------------------

Begin
  Date (today);                  { <---- Get date from system    }
  Assign (glmaster, glname);     { <---- Assign DOS file name    }
  Rewrite (glmaster);            { <---- Open for writing        }
```

```
{
-------------------------------------------------------------------
        Initialize the static fields in the master record.
-------------------------------------------------------------------

        With master Do          { same for every record }
            [
            last_activ := today;
            last_tran := Ø;
            curr_debit := Ø;
            curr_credit := Ø;

            For index := 1 to hist_size Do
                [
                hist_debits[index] := Ø;
                hist_credits[index] := Ø;
                ];
            ];

-------------------------------------------------------------------
                Input the first account number.
-------------------------------------------------------------------

        Writeln;
        Write ('Acct number ------> ');         Readln (acct_in);
{
-------------------------------------------------------------------
      Input the user defined fields and create the record.
-------------------------------------------------------------------
{
        While acct_in <> end_input Do
            [
            With master Do
                [
                acct_no := acct_in;
                Write ('Acct type --------> '); Readln (acct_type);
                Write ('Description ------> '); Readln (acct_desc);
                Write ('Begin balance ----> '); Readln (beg_bal);
                Write ('Ending balance ---> '); Readln (end_bal);
                ];

            glmaster^ := master;{ <---- Assign buffer variable  }
            Put (glmaster);     { <---- Put a component in file }
{
-------------------------------------------------------------------
                Input the next account number.
-------------------------------------------------------------------
{
            Writeln;
            Write ('Acct number ------> ');     Readln (acct_in);
            ];
```

```
        Program end   Don't forget to CLOSE the file
-------------------------------------------------------------------

      Close(glmaster);

End.
```

Figure 6-10. GLLOAD.PAS General Ledger File Loading Program

We use the ASSIGN statement to connect the Pascal file name "glmaster" to the actual DOS file name "glmaster.dat." The REWRITE statement immediately follows. This opens the file for writing or output. Anything that may previously have been in the file becomes inaccessible after the REWRITE is executed.

The next section of the program is a WITH structure that initializes all of the static fields in the record variable "master." These fields will always be the same in every record.

After prompting for and inputting the initial account number from the keyboard, the program enters a WHILE structure: it will remain in this structure as long as the account number entered is not zero.

One by one, the user-definable fields in the G/L master record are prompted for and input. This is all done using the predeclared procedures WRITE and READLN. (We're already familiar with using these for I/O with the console. Later in this chapter we will see how they can be used in a more general way.)

Finally, the record variable "master" is assigned to the buffer variable "glmaster^" and the PUT procedure is called. This literally writes out all of the bytes that are contained in the buffer variable. The DOS does some internal "housekeeping" to keep track of the disk file. Each time PUT is executed, it moves a pointer so that the next record will be written in the next available space on the disk.

All it takes to end this program is entering an account number of zero. The program will then CLOSE the file. This is done automatically in most cases by the Pascal termination module, but it is still a good idea to include it as an explicit program statement.

Listing the General Ledger File

Now we need some way of listing the accounts we have just loaded into the General Ledger file. Since the records are stored in a BINARY type file, we can't use the TYPE command in the DOS to list them. We'll

take this opportunity to demonstrate how to use the printer as a SEQUENTIAL file of type TEXT. (See Figure 6-11.)

Figure 6-11. General ledger file listing program

```
{*****************************************************************
GLLIST.PAS      Print General Ledger File Listing
-----------------------------------------------------------------
        This program is used to print a listing of the General
Ledger file.  The printer is treated just like another sequential
file.
-----------------------------------------------------------------}

Program Gllist (input,output);

Const
  glname = 'glmaster.dat';      { <---- DOS file name           }
  hist_size = 12;

{$include:'glmaster.rec'}       { <---- Compile TYPE declaration }

Var
  glmaster      :file of gl_master;     { <---- FILE declaration }

  printer       :text;          { <---- Output file is ASCII    }

{-----------------------------------------------------------------
        Main program starts here.  Initialize the files
-----------------------------------------------------------------}

Begin
  Assign (glmaster, glname);    { <---- G/L file                }
  Reset (glmaster);

  Assign (printer, 'PRN');      { <---- System printer          }
  Rewrite (printer);

{-----------------------------------------------------------------
               Print from the buffer variable
-----------------------------------------------------------------}

        While not eof(glmaster) Do      { <---- loop through file }
            |
            With glmaster^ Do           { <---- Output to printer }
                Writeln (printer,
                         acct_no,
                         acct_desc      :22,
                         last_activ     :10,
                         last_tran      :10:2,
```

```
                curr_debit      :10:2,
                curr_credit     :10:2
                );

            Writeln (printer);      { <---- Double spacing  }
            Get (glmaster); { <---- Get the next record     }
            };

End.
```

Figure 6-11. GLLIST.PAS General ledger File Listing Program

Again, notice the $INCLUDE compiler metacommand that will cause the RECORD declaration to be compiled in line. The files that are declared are "glmaster" and "printer." The "printer" file is declared as a TEXT file, so that we can use the standard procedure WRITELN to output lines of characters.

The ASSIGN statements specify the DOS file name for the General Ledger file and the special file "PRN" for the printer. Notice the RESET for the file being read, and the REWRITE for the file being written.

The RESET also performs the first GET of a record component in the file "glmaster." That's why we can enter the WITH structure and begin printing the fields.

There is something different about the WRITELN statement here. It has the file name "printer" as the first parameter. Whenever the first parameter is a FILE type, the routine will output to that file instead of to the display.

Then, one by one, the fields we wish to print are output in the same WRITELN statement. Notice the formatting that can be done by using specifications following each variable name. This can allow you to space your printouts however you want.

A second WRITELN to the file "printer" achieves double spacing. Then, the next record in the file is retrieved with the GET procedure. If there are no more records in the file, the result of the GET will set EOF (glmaster) equal to TRUE, causing termination of the WHILE structure and the end of the program.

An example of the printout from the "Gllist" program, shown in Figure 6-11, is set forth in Table 6-1.

Table 6-1. General ledger file listing

```
51Ø  Consulting Income      Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
52Ø  Royalty Income         Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
59Ø  Other Income           Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
61Ø  Owners Draw            Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
52Ø  Travel Expense         Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
53Ø  Automotive Expense     Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
54Ø  Other Expense          Ø7-17-83      Ø.ØØ      Ø.ØØ      Ø.ØØ
```

This type of general ledger can be very useful in managing the budget at home or for a small professional business. Later in this chapter we will show you how to update the General Ledger with the transactions of your business. For now, we'll demonstrate the use of TEXT type files for a similar application: a Customer file.

The Customer File

In addition to a General Ledger with accounts for income and expense, it is often useful to have a file of the customers of a business. The same principle could be applied to creditors. We will make this file a TEXT type file since many of the fields involved are STRING type. This will also make the file accessible to the TYPE command in the DOS and to most word processing programs.

We will also include some REAL type variables in our Customer file. These are to keep track of customer balances and year-to-date activity. Remember, when these are handled by the TEXT file routines READLN and WRITELN, they are converted to a string of characters. The default format for REAL is the scientific notation form with a mantissa and exponent. Since that is hard to read, we'll have to specify the format in the WRITELN statement that creates the records.

Customer Field Declarations

The fields are declared in a program segment that can be $INCLUDEd in any compilation. Since the Customer file will be a TEXT file, we do not GET and PUT the RECORD type variable as we did with the General Ledger file. Instead, we will use READLN and WRITELN to read and write a string of characters that represent the fields converted to an ASCII form. So, instead of declaring a RECORD type, we will declare each of the variables that we want to be part of the Customer file record on the disk. Each of these fields will be formatted by the WRITELN procedure. Figure 6-12 shows a file that contains our VAR declarations.

Figure 6-12. Customer file field declarations

```
{*****************************************************************
CUMASTER.REC    Customer Master Field Declarations
-----------------------------------------------------------------

Var
  acct_no       :integer;       { Account number for reference  }
  name,                         { Customer name                 }
  address,                      { Street address                }
  city_sta                      { City and State                }
                :string(25);
  zipcode       :string(9);     { Zipcode                       }
  areacode      :string(3);     { Area code                     }

  telephone,                    { Telephone number XXX-XXXX     }
  date_opened                   { Date account opened MM/DD/YY  }
                :string(8);
  beg_balance,                  { Beginning A/R balance         }
  ytd_charges,                  { Year-to-date charges          }
  ytd_payments,                 { Year-to-date payments         }
  end_balance                   { Ending A/R balance            }
                :real;

  status        :char;          { Account status                }

-----------------------------------------------------------------
Figure 6-12.     CUMASTER  Customer File Field Declarations
*****************************************************************}
```

We'll use some typical fields. An account number will provide easy reference. We'll also want the name and address (city, state, and zip code). It's a good idea to have the area code and telephone number, and let's throw in the date that the account record was created on the disk. So far, all of these fields are STRING type. Let's add four REAL types to keep track of the account balance and activity, and for good measure, one CHARacter type as a status indicator.

Loading the Customer File

No doubt you will notice that loading the Customer file is very similar to loading the General Ledger file. In this program, set forth in Figure 6-13, you can enter every field except the "date_opened" field.

The declarations for this program consist most importantly of the $INCLUDE compiler metacommand that will compile the field declarations for the Customer file. After that, the Pascal file variable "cumaster" is declared as a TEXT type (ASCII structure file).

Figure 6-13. Customer file load program

```
{*****************************************************************
CUSTLOAD.PAS    Create Customer Master File
-----------------------------------------------------------------
        This program creates the Customer master file from
data entered by the user from the keyboard.  All fields except
the "date_opened" are defined by the user.  This is a TEXT file.
-----------------------------------------------------------------
}
Program Custload (input,output);

Const
        end_input = Ø;

{$include:'cumaster.rec'}       { <---- Customer field declarations }

Var
        cumaster        :text;  { <---- Customer FILE declaration }

Value
        status := 'n';
{
-----------------------------------------------------------------
                    External date procedure.
-----------------------------------------------------------------
}
Procedure Date (Var s:string);          external;
{
-----------------------------------------------------------------
        Main program begins here.  The DOS file name is entered
upon program execution using the READFN procedure.
-----------------------------------------------------------------
}
Begin
  Date (date_opened);               { <---- Initialize the date   }
  Write ('Enter file name --> '); { <---- Prompt for file name  }
  Readfn (input, cumaster);         { <---- Input from keyboard   }
  Rewrite (cumaster);               { <---- Open the file         }
{
-----------------------------------------------------------------
                    Input the first account.
-----------------------------------------------------------------
}
        Readln;
        Write ('Acct number --> ');     Readln (acct_no);
```

```
{-------------------------------------------------------------------
                Input all of the customer fields
-------------------------------------------------------------------}
!
    While acct_no <> end_input Do
        {
        Write ('Name ---------> ');     Readln (name);
        Write ('Address ------> ');     Readln (address);
        Write ('City/State ---> ');     Readln (city_sta);
        Write ('Zipcode ------> ');     Readln (zipcode);
        Write ('Area code ----> ');     Readln (areacode);
        Write ('Telephone ----> ');     Readln (telephone);
        Write ('Begining bal -> ');     Readln (beg_balance);
        Write ('YTD charges --> ');     Readln (ytd_charges);
        Write ('YTD payments -> ');     Readln (ytd_payments);
        Write ('Ending bal ---> ');     Readln (end_balance);

{$include:'cumaster.wln'}       { <---- WRITELN for cust. file  }
!
{-------------------------------------------------------------------
                    Input the next account.
-------------------------------------------------------------------}
!
        Writeln;
        Write ('Acct number --> ');     Readln (acct_no);
        ];
!
{-------------------------------------------------------------------
          Program end.  Don't forget to CLOSE the file.
-------------------------------------------------------------------}
!
        Close (cumaster);       { <---- CLOSE the file          }
End.
|
{-------------------------------------------------------------------
Figure 6-13.     CUSTLOAD.PAS  Customer File Load Program
*******************************************************************}
```

The EXTERNAL procedure DATE is used, this time to initialize the variable "date_opened."

In this program, we will use the READFN procedure to get the actual DOS file name. This is an alternative to the ASSIGN statement which assigns the same name each time. When we use the READFN procedure, we can create several different Customer files with different names. Once the file name has been resolved, the REWRITE statement initializes the file for output.

There is a dummy READLN statement here to clear the line feed

character from the buffer variable of the file "input" before attempting any further READLN calls. We had to pay close attention to the *IBM Personal Computer Pascal Compiler* manual to catch this one: see the explanation of READFN on page 12-31 of the first edition (August 1981).

One by one, the fields for the Customer file record are prompted and input using the same WRITE/READLN combination. Then comes an important twist to using this TEXT file method: we must be very careful that each time we write records to the Customer file from this or any other program, we use exactly the same format. For this reason, we create another Pascal code segment that consists of only a WRITELN statement with all the fields in the correct order, as shown in Figure 6-14.

Figure 6-14. WRITELN for customer file

```
{*************************************************************
CUMASTER.WLN    Customer Master File Writeln Statement
-------------------------------------------------------------
}
        Writeln (cumaster,
                        acct_no,
                        name,
                        address,
                        city_sta,
                        zipcode,
                        areacode,
                        telephone,
                        date_opened,
                        beg_balance     :10:2,
                        ytd_charges     :10:2,
                        ytd_payments    :10:2,
                        end_balance     :10:2
                );

-------------------------------------------------------------
Figure 6-14.    CUMASTER.WLN  WRITELN for Customer File
*************************************************************}
```

The STRING variable types are written out consecutively, one after another. The REAL types, however, are formatted. The 10:2 following each REAL type causes the WRITELN procedure to convert the REAL to a string of 10 characters. The string includes a decimal point, and has two digits to the right of the decimal.

The field size specified for numeric output should be large enough to guarantee that at least one blank will be generated by the formatting, so that each field is separated from its neighbor by a blank. That's the only

way that the READLN procedure can tell where one field ends and the next begins.

As you might expect, there is also a corresponding Pascal code segment containing nothing but a READLN call. Notice in Figure 6-15 that we do not need any formatting here.

Figure 6-15. READLN for customer file

```
{*****************************************************************
CUMASTER.RLN     Customer Master File Readln Statement
-----------------------------------------------------------------

        Readln (cumaster,
                        acct_no,
                        name,
                        address,
                        city_sta,
                        zipcode,
                        areacode,
                        telephone,
                        date_opened,
                        beg_balance,
                        ytd_charges,
                        ytd_payments,
                        end_balance
                );
}
-----------------------------------------------------------------
Figure 6-15.      CUMASTER.RLN  READLN for Customer File
*****************************************************************}
```

Listing the Customer File

Once we have loaded our Customer file, we will probably want to print it out in some form. Since it is a TEXT file, there are several techniques available. We can read the file into our word processor, and print it out from there. Or we can use the DOS command TYPE to display the file on the screen or printer (using the Ctrl PrtSc key). Or, more interestingly, we can write a Pascal program to print out the file, as we have in Figure 6-16. We'll call this program "Custlist".

Once again, we'll $INCLUDE the field declarations in the file "cumaster.rec" in the compilation. As in the General Ledger printing program, the disk file is ASSIGNed and RESET. The file "printer" is assigned to the special device "PRN" and initialized for printing with the REWRITE procedure.

Figure 6-16. Customer file listing program

```
{*************************************************************}
CUSTLIST.PAS    Print Customer File Listing
-------------------------------------------------------------
        This program prints a listing of the Customer master
file.  Not all of the fields are printed.
-------------------------------------------------------------
}
Program Custlist (input,output);

Const
  cuname = 'cumaster.dat';       { <---- DOS file name          }

{$include:'cumaster.rec'}        { <---- Compile field declaration }

Var
  cumaster      :text;           { <---- Customer file          }

  printer       :text;           { <---- Printer file           }
{
-------------------------------------------------------------
         Main program begins here.  Initialize files.
-------------------------------------------------------------
}
Begin
  Assign (cumaster, cuname);     { <---- Customer file is on disk }
  Reset (cumaster);

  Assign (printer, 'PRN');       { <---- Assign printer device}
  Rewrite (printer);
{
-------------------------------------------------------------
             Read a record and print sequentially.
-------------------------------------------------------------
}
    While not eof(cumaster) Do
      {
{$include:'cumaster.rln'}        { <---- Compile READLN call    }

       Writeln (printer,
                name,
                ' (', areacode, ')',
                telephone       :9,     { <---- Length override }
                date_opened     :10
                );

       Writeln (printer,
                address);
```

```
        Writeln (printer,
                city_sta, zipcode,
                beg_balance     :10:2,  { <---- Format override }
                end_balance     :10:2
                );

        Writeln (printer);      { <---- Double spacing          }
        ];
End.
```

Figure 6-16. CUSTLIST.PAS Customer File Listing Program

The program enters a WHILE structure, waiting for the Boolean function EOF to indicate that the end-of-file has been reached. Here, notice the $INCLUDE of the READLN call. This is very important, and ensures consistency between different programs that access the Customer file.

We don't want all of the fields on this printout. We just want the name and address in the standard three-line format used for addressing, and the REAL types that contain the account balances.

Several WRITELN calls perform the printing. A dummy WRITELN to the file "printer" causes double spacing between records. Table 6-2 gives an example of the Customer File listing.

Table 6-2. Customer file listing

```
L. A. Herald Examiner      (213) 744-8085  07-15-83
1111 S. Hill Street
Los Angeles, CA          90015          0.00   1400.00

Independent-Journal        (415) 883-8600  07-15-83
7100 Alameda Del Prado
Novato, CA               94947       4500.00  26540.00

The Waite Group            (415) 459-3830  07-15-83
1505 Fifth Avenue
San Rafael, CA           94901          0.00      0.00
```

DIRECT Mode Files

Thus far, we have dealt with files as though they were pipelines through which all the data required by a program travels in sequence. As we've seen, the SEQUENTIAL file mode is practical for many applications. However, more sophisticated computer applications must be able to access data randomly and for that, we need to use the DIRECT mode.

Setting DIRECT Mode

Before a file can be treated as a DIRECT access file, the Pascal program must change the MODE indicator for the file. This changes a field within the "File Control Block." (The File Control Block is used by the DOS to hold data about each file being accessed by the system.) Since the File Control Block is a Pascal RECORD type, we can use the file identifier and set the mode in an assignment statement. This must be done before the file is opened with RESET or REWRITE. For example:

```
filename.mode := direct;
Reset (filename);
```

Opening a Direct Mode File

The procedures RESET and REWRITE, which are used in Pascal to open files, operate slightly differently on DIRECT mode files than on SEQUENTIAL or TERMINAL mode files. Either RESET or REWRITE will open a file for both reading and writing operations in DIRECT mode. However, RESET will cause an error if the DOS file name specified can not be found, whereas REWRITE will create a new file if the name is not found.

The SEEK Procedure

To access a file in the DIRECT mode, we need to supply the *record number* of the component of the file we wish to access. Remember, files can be thought of as large arrays, each component having a given number. So DIRECT access allows you to read or write any record in the file without having to traverse the file sequentially. The SEEK procedure is used just prior to a GET, PUT, READ, READLN, WRITE, or WRITELN. It looks like this:

```
Seek (filename, record_no);
Get (filename);
```

```
Seek (filename, record_no);
Write (filename, data);
```

Building an Index Table

In order to have the record number to pass to the SEEK procedure, we have to know what each record contains, and where to look for a specified account. A common solution to this problem is to build an *index table* in memory at the beginning of the program, and then use it to determine the record numbers needed for SEEK.

We will build an index table very soon. For now, you should know that because the index table is in memory, it can be searched, even sequentially, in very little time compared to actually searching through the disk file. The entire file is traversed once, at the beginning of the program, to build the index table. But after that, random access to the disk is easy. This method works quite well for files up to about 100 records on a mini-diskette. For larger files, the time required to build the index table each time the file is accessed will start to become unacceptable. Then, the index table should be maintained as a separate file on the disk.

Remember the File Pointer

The SEEK procedure doesn't actually do anything to the file. It just calculates where the requested record is located, and sets the file pointer to that record. The next read or write operation will then access the correct record. However, remember that the file pointer is *advanced* by either a read or write operation. This must be considered when reading or writing in the same file. You must use the SEEK procedure before both the GET and the PUT procedures to make sure you are accessing the correct record.

Updating Files

In both of the examples we've developed in this chapter, there is an obvious need to be able to update a file once it has been written. In fact, that's the whole idea behind setting up files like these. Typically entries in them will change quite frequently.

Several different methods are used to update disk files, depending upon the nature and frequency of the transactions. When there are only a few transactions, you may wish to save them up, and enter them all at once, say at the end of a month. But if there are a lot of transactions, you

may want to perform your data entry more frequently. Sometimes it is necessary to update a file "on the fly" so that every transaction is immediately added to the master file.

Immediate or Batch Update

The two updating techniques in common use for all kinds of business data processing are the *immediate* and the *batch* update. Simply put, an immediate update changes the file as soon as the transaction has been entered. The batch update, on the other hand, involves an intermediate "batch file" into which the transactions are loaded when they are entered. Then the batch file is usually listed, and proofed to make sure no entry errors were made. Only after the listing is found to be correct will the transactions in the batch file be used to update the master file.

Our example programs are designed to use the simpler, immediate update technique because all we want to do is illustrate the use of DIRECT mode file access, but you should be aware of the other option. The batch update approach is definitely superior for accuracy and documentation. When you need to update several files for each transaction, the batch method will speed up your data entry.

Updating the Customer File

The Customer file example is still fresh in our minds, so let's tackle a program to update it with transactions. Since we are updating in immediate mode, we'll enter the transactions from the keyboard, so they will update the file right away. We'll need a program that accepts keyboard input, and updates a disk file. We would like to use the customer's account number as a reference for the transactions, and as a means of accessing the file. Therefore, we will need an index table that contains entries matching account numbers with record numbers. Then, when a specific account is sought, the program can search the index table to find the record number.

Remember, whenever we want to access a TEXT file in DIRECT mode, we have to specify the number of bytes that will be read for each component. In this case, the length of all of the fields that make up the customer record is 151 bytes. The length is declared with the Pascal file variable "cumaster."

As you can see in Figure 6-17 the Customer file update program has the usual declarations at the beginning, including a type declaration which needs careful examination. The "index entries" are described here as RECORD types consisting of an INTEGER type account number, and a WORD type record number. Next, we see the expected $INCLUDE used to bring in the Customer file field declarations, and some work

variables. Finally, the "index" table is declared as an array of the index entry records.

Figure 6-17. Customer file update program

```
{*****************************************************************}
CUSTUPDT.PAS            Customer File Update Program
{-----------------------------------------------------------------}
       This program updates the Customer file with
transactions entered by the user from the keyboard.  The
update is immediate, and no validation is performed.
{-----------------------------------------------------------------}

Program Custupdt (input,output);

Const
  cuname = 'cumaster.dat';      { <---- DOS file name          }
  max_recs = 10;                { <---- Maximum record count   }
  end_input = 0;

Type
  index_entry = record          { <---- Index table TYPE       }
    acct       :integer;
    rec_no     :word;
  end;

Var
  cumaster     :text (151);     { <---- Customer file with length }

{$include:'cumaster.rec'}       { <---- Customer field declarations }

Var
       amount          :real;

       acct_in         :integer;

       entry,
       rec_count,
       rec_addr        :word;

       index           :array[1..max_recs] of index_entry;

{-----------------------------------------------------------------}
        Main program begins here   Initialize the file.
{-----------------------------------------------------------------}

Begin
  Assign (cumaster, cuname);    { <---- Customer file          }
  cumaster.mode := direct;      { <---- Set DIRECT mode        }
  Reset (cumaster);
```

```
{
-------------------------------------------------------------------
                    Build an index table
-------------------------------------------------------------------
}
        rec_count := Ø;
        While not Eof (cumaster) Do
                [
                rec_count := rec_count + 1;
                With index[rec_count] Do
                        [
                        Readln (cumaster, acct);
                        rec_no := rec_count;
                        ];
                ];
{
-------------------------------------------------------------------
                 Input the first account number.
-------------------------------------------------------------------
}
        Write ('Acct number -------> ');        Readln (acct_in);
{
-------------------------------------------------------------------
             Look up the account in the index table.
-------------------------------------------------------------------
}
        While acct_in <> end_input Do
                [
                rec_addr := Ø;
                For entry := 1 to rec_count Do
                    With index[entry] Do
                        If acct = acct_in
                            Then [
                                rec_addr := rec_no;
                                Break;
                                ];
{
-------------------------------------------------------------------
            SEEK the record if account numbers match.
-------------------------------------------------------------------
}
                If rec_addr > Ø
                    Then [
                        Seek (cumaster, rec_addr);

{$include:'cumaster.rln'}       { <---- Customer file READLN call }
```

```
-------------------------------------------------------------------
              Display the name, and input amount
-------------------------------------------------------------------

                        Writeln (name, '(record', rec_addr, ')' );
                        Write ('Enter amount ------> ');
                        Readln (amount);

-------------------------------------------------------------------
             Assign the amount based upon its sign.
-------------------------------------------------------------------

                        If amount > Ø
                            Then ytd_charges :=
                                ytd_charges + amount
                            Else ytd_payments :=
                                ytd_payments + amount;
;
-------------------------------------------------------------------
           Now, SEEK again and write back the record.
-------------------------------------------------------------------

                        Seek (cumaster, rec_addr);

{$include:'cumaster.wln'}       { <---- Customer file WRITELN call }

                        ]
;
-------------------------------------------------------------------
            Error message if no account number match.
-------------------------------------------------------------------
}
                    Else
                        Writeln ('Invalid account number');
;
-------------------------------------------------------------------
                 Input the next account number
-------------------------------------------------------------------

                Writeln;
                Write ('Acct number -------> ');
                Readln (acct_in);
        ].
```

```
;
{-------------------------------------------------------------}
                Program end.  CLOSE the file.
{-------------------------------------------------------------}
      Close (cumaster);
End.
;
{-------------------------------------------------------------}
```

Figure 6-17. CUSTUPDT.PAS Customer File Update Program

To use the file, we ASSIGN the DOS file name, set the DIRECT mode, and open the file using RESET.

Next, we build the index table. The WHILE structure causes the program to traverse the entire file sequentially. The records are counted as we go along. Notice the READLN call that only reads the "acct" number (which has to be the first field in the record). One by one, the index table entries are created so that the file may be accessed by the account numbers.

The data entry for this program consists of the account number and the transaction amount. As long as the account number is not zero, the program will continue processing transactions.

After getting the desired account number, the program searches the index table for the account number entered. This is done in a FOR structure, since we know the number of records in the file. One by one, each index entry is compared with the account number entered from the keyboard. If a match is found, the BREAK command is executed, dropping the program out of the search loop.

Once out of the loop, the program must determine whether the account number was found. This is easy since the variable "rec_addr" is set to zero each time before the index table is searched. If a matching entry is not found in the index table, the program will come out of the loop with "rec_addr" still equal to zero.

The rest of the program is inside the IF structure. A positive value for "rec_addr" will be the expected path. The SEEK procedure is called, with the record number determined from the index table. Then we use the $INCLUDE for the READLN statement to read the specified record.

For verification, the customer's name is displayed, along with the record number, before the transaction is entered. If you select the wrong customer by mistake, enter a transaction amount of zero to avoid affecting the file.

The amount entered is examined to determine whether to update the charges field or payments field in the customer record. Once the arithmetic update has been completed, the SEEK procedure is again

called to prepare for writing the customer record back into the file. As before, this is accomplished by the $INCLUDE command, to compile the WRITELN statement for the Customer file.

The program will continue accepting input until a zero account number is entered.

Updating the General Ledger File

As another example of using DIRECT mode files, let's develop an update program for the General Ledger file we created earlier in this chapter. Basically, the update philosophy will be the same as for the Customer file. The accounts will be accessed by using an index table, constructed from their account numbers and record numbers. The difference between this program and the Customer file update program we just finished is that we'll use GET and PUT, instead of READLN and WRITELN, since this is a BINARY type file.

This program, shown in Figure 6-18, follows the same approach as used in the Customer file update program. The index table is built almost exactly the same. How convenient! Note, however, that the index table is built in a WHILE structure as long as EOF remains FALSE. But, the first time into the WHILE, there is already a record in the buffer variable because RESET does an initial GET from the file. That's why you see the GET statement as the last thing in the WHILE structure.

Figure 6-18. General ledger file update program

```
{*************************************************************
GLUPDATE.PAS            General Ledger File Update
-------------------------------------------------------------
        This program updates the General Ledger file with
transactions entered by the user from the keyboard.  The update
is immediate.
-------------------------------------------------------------}

Program Glupdate (input,output);

Const
  glname = 'glmaster.dat';       { <---- DOS file name          }
  max_recs = 10;
  end_input = 0;
  hist_size = 12;

{$include:'glmaster.rec'}        { <---- G/L record declaration }

Type
  index_entry = record           { <---- Index table TYPE       }
    acct        :integer;
```

```
    rec_no      :word;
end;

Var
  glmaster      :file of gl_master;     { <---- G/L file        }

  work_record   :gl_master;     { <---- Update work area        }

Var
        amount          :real;

        entry,
        rec_count,
        acct_in         :integer;

        rec_addr        :word;

        index           :array[1..max_recs] of index_entry;

        today           :string(8);

{
-------------------------------------------------------------------
                Date procedure is external.
-------------------------------------------------------------------
}
Procedure Date (Var s:string);  external;

{
-------------------------------------------------------------------
     Main program begins here.  Initialize date and files.
-------------------------------------------------------------------
}
Begin
  Date (today);
  Assign (glmaster, glname);
  Reset (glmaster);
{
-------------------------------------------------------------------
                    Build an index table
-------------------------------------------------------------------
}
        rec_count := Ø;
        While not Eof (glmaster) Do
                [
                work_record := glmaster^;
                rec_count := rec_count + 1;
                With index[rec_count] Do
                        |
                        With work_record Do
                            acct := acct_no;
                        rec_no := Wrd (rec_count);
```

```
                ].
            Get (glmaster);
            ];

{-------------------------------------------------------------}
            Set DIRECT mode and RESET the file
{-------------------------------------------------------------}

    glmaster.mode := direct;
    Reset (glmaster);

{-------------------------------------------------------------}
                Enter first account number.
{-------------------------------------------------------------}

    Write ('Acct number -------> ');
    Readln (acct_in);

{-------------------------------------------------------------}
          Look up the account in the index table
                to get the record number.
{-------------------------------------------------------------}

    While acct_in <> end_input Do
            [
            rec_addr := Ø;
            For entry := 1 to rec_count Do
                With index[entry] Do
                    If acct = acct_in
                        Then [
                            rec_addr := rec_no;
                            Break;
                            ];

{-------------------------------------------------------------}
            If found, SEEK and GET the record.
{-------------------------------------------------------------}

            If rec_addr > Ø
                Then [
                    Seek (glmaster, rec_addr);
                    Get (glmaster);
                    work_record := glmaster^;
```

```
{-------------------------------------------------------------}
        Display the account and update the fields here.
{-------------------------------------------------------------}

                    With work_record Do
                        {
                        Writeln (acct_desc, '(record', rec_addr, ')' );
                        Write ('Enter amount ------> ');
                        Readln (amount);

                        last_tran := amount;
                        last_activ := today;

                        If amount > Ø
                            Then curr_debit :=
                                curr_debit + amount
                            Else curr_credit :=
                                curr_credit + amount;
                        };

{-------------------------------------------------------------}
            Now, write the record back to the file.
{-------------------------------------------------------------}

                    glmaster^ := work_record;
                    Seek (glmaster, rec_addr);
                    Put (glmaster);
                    }

{-------------------------------------------------------------}
            Error message if account is not found.
{-------------------------------------------------------------}
;
                Else
                    Writeln ('Invalid account number');
;
{-------------------------------------------------------------}
                Enter the next account number.
{-------------------------------------------------------------}

            Writeln;
            Write ('Acct number -------> ');
            Readln (acct_in);
            };
```

```
{
-------------------------------------------------------------------
                Program end.  CLOSE the file.
-------------------------------------------------------------------
}
      Close (glmaster);
End.
{
-------------------------------------------------------------------
```

Figure 6-18. GLUPDATE.PAS General Ledger File Update Program

Once the index table is ready, the program begins the data entry and update section. This is all contained in another WHILE structure which operates as long as the account number entered is not zero.

For each transaction entered, the program searches the index table for a matching account number. If found, the SEEK and GET procedures are used to retrieve the record. We chose to update several fields in the General Ledger file, including the last transaction date.

Finally, notice that replacing the record in the file requires both another SEEK and a PUT since the preceding GET call advanced the file pointer to the next record.

Summary

You've seen that it's possible to use a file for many different types of applications. Now, all of the major features of Pascal have been discussed. We can save information on the disk for future use, and we can pass data back and forth between the program and the real world, all using files.

Exercises

1. How many components can a file have? How many components can be used at a time?

2. What Pascal statements are needed to open the file "READ.ME" for input?

3. What is the difference between binary and TEXT (or ASCII) files?

4. How is PUT different from WRITE for output to a file?

5. What is direct access on a file?

Solutions

1. A file can have as many components as will fit on the diskette. Only one component can be used at a time.

2. To open the file "READ.ME" for input, you need: ASSIGN (file_var, "READ.ME"); RESET (file_var);

3. Binary files contain data just as it is stored in memory; TEXT (or ASCII) files translate all data into a form that can be read by people.

4. PUT outputs one component of a binary file; WRITE translates as many variables as specified into ASCII strings and outputs them to a TEXT file.

5. Direct access is the ability to extract a component of a file from the middle without having to read through from the first component.

7

Systems of Programs

Concepts

- Determining when a system is needed
- The system library
- Top-down and modular programs
- Programs, modules, and implementations
- Source and object code files
- Linking object files
- Units and Interfaces
- Assembly language routines for graphics
- Color graphics

Keywords

EXTERNAL, PUBLIC, MODULE, INTERFACE, IMPLEMENTATION, UNIT, USES, $INCLUDE, F.ERRS, F.TRAP

So far, the examples in this book have been relatively small programs that perform an entire function by themselves. However, it is often useful to write several programs that can work together and perform more complex tasks. Actually, we've already used a few EXTERNAL procedures and functions. In chapter 5, we used the assembly language routines PORTIN and PORTOUT to experiment with the monochrome display and the sound chip. Those two routines and the Pascal program which called them constitute a simple *system of programs*.

However, there are other, more powerful ways that programs can be linked together. For instance, IBM Pascal allows us to put together a library of procedures and functions which are usable by other programs. We can add new procedures and functions by writing a MODULE. When using a complex system of interrelated declarations we can reduce the possibility of error by using UNITs and IMPLEMENTATIONs.

We'll discuss some relatively advanced techniques in this chapter. If

you are just learning to program, you might want to skip this chapter for the time being. If you are interested in using only a few MODULEs, then just read the first third or so of the chapter. But if you already are, or

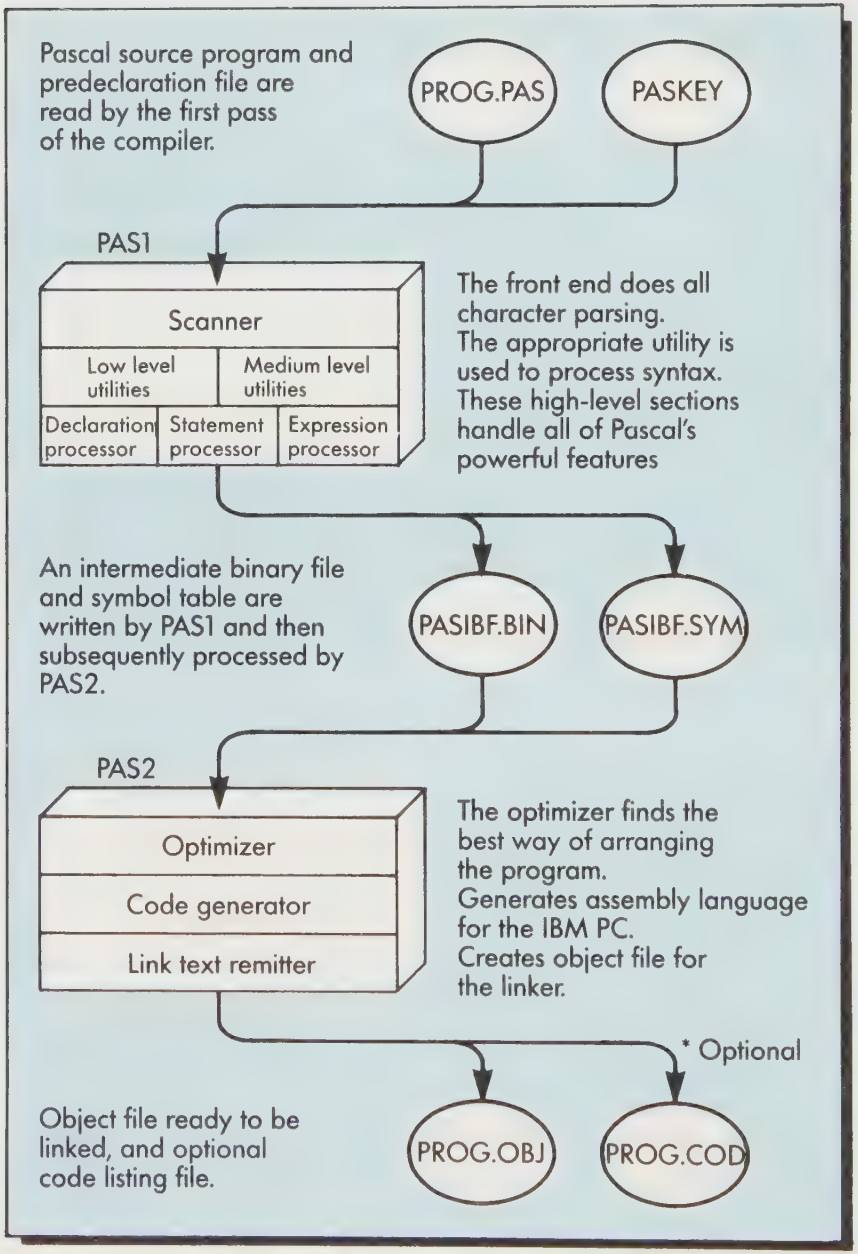


Figure 7-1. Pascal compiler system of programs

intend to be, an advanced programmer who fully utilizes the abilities of the Pascal compiler, the final two-thirds of this chapter are for you.

What Is a System of Programs?

There are endless possibilities for using systems of programs. Figure 7-1 illustrates how the Pascal compiler itself is a system of several modules incorporated into two programs, PAS1 and PAS2.

The first program, PAS1, contains several sections. These are all related to *parsing* the Pascal source program statements into the intermediate code, ready for PAS2. The second program, PAS2, takes over where PAS1 left off. It completes the compilation process, and generates an object file, ready for the linker program.

There are many other situations which require a system of programs. In business, a system might perform some complex task such as tracking sales or preparing an income tax form. At home, you might have a collection of procedures and functions to help in writing future programs. Or, you might purchase a system of special routines to help with such complex tasks as graphics or keyed file access.

When Should a System Be Used?

In general, any time you are solving a large, complex problem you should consider breaking it down into smaller parts by using a system of programs. Splitting the problem into separate parts involves some extra work, but it can be well worth the effort. Reuse of commonly used procedures can save time when writing and compiling programs and reduce the possibility of typing errors. Finding errors in a large, complex program can be very difficult. Writing small portions of the program and testing them separately can greatly reduce the debugging time for the final program.

Sometimes it is necessary to split a large program into several parts just to make each part fit in a limited amount of memory. The Pascal compiler, for instance, was divided into two passes so that it could handle large programs even on systems with minimal memory.

Top-Down and Modular Programming

There are two terms used quite often by professional programmers relating to writing systems of programs: *top-down* and *modular* programming. The top-down approach involves dividing a complex problem into several simpler problems, then dividing each of those further into step-by-step processes, and so on until the problem is

completely solved. Imagine for example, that you were building a house. You would first lay out the floor plan, then decide where to put the cabinets, and later decide on the number of shelves in each cabinet. In a similar way, top-down programming goes from the large, general problem to a number of specific, detailed problems.

Modular programming uses several small components, called *routines*, to construct a program; each routine performs a single, well-defined function. In a business system, one routine might be used for reading the customer file and another used to sort data to be printed on a report.

IBM Pascal provides a number of features to encourage the top-down approach to problem solving and to facilitate the use of modular programming. These features, which we'll describe in the next sections, are expansions of the PROGRAM structure we have been using so far.

PROGRAMs, MODULEs, and IMPLEMENTATIONs

The Pascal compiler accepts source files in three different divisions: PROGRAMs, MODULEs, and IMPLEMENTATIONs. All divisions are source files which are compiled into object files by the Pascal compiler. IBM calls these divisions *Compilands*. A chapter in the *IBM Personal Computer Pascal Compiler* manual covers this topic; don't be discouraged by its description of compilands. They aren't really that complicated, as we'll see. A division must start with one of the division identifiers: PROGRAM, MODULE, or IMPLEMENTATION OF. All divisions terminate with the keyword END and a period. When compiled, each division produces a single object file to be used by the linker.

The PROGRAM Division

A PROGRAM file is the division containing the body of a main program. Every executable file produced by the linker contains the object file from exactly one PROGRAM division. The PROGRAM division contains the first statement performed when the program is run (after initialization). The program object file is the first name supplied to the "Modules:" prompt while linking the program. For example,

```
IBM Personal Computer Linker
Version 1.1Ø (C)Copyright IBM CORP 1982
Object Modules [.OBJ]: Progname+Modname1+Modname2+...
```

The MODULE Division

A MODULE file is a division that only contains definitions of PUBLIC procedures, variables, or functions. As we have seen, the Pascal keyword PUBLIC is an attribute that can be assigned to a variable, function, or procedure. It means that the variable, function, or procedure will be available to any PROGRAM that uses the MODULE.

A MODULE does not have a BEGIN...END block outside of any procedures or functions declared by it. A PROGRAM and any number of MODULEs may be compiled separately and combined with the linker to form a single executable program. (We'll see an example of this soon, when we develop the "Stopwtch" PROGRAM and link it with a MODULE called "Prnttime.")

The IMPLEMENTATION Division

The IMPLEMENTATION division contains the statements that define the procedures and functions declared in a special file called an INTERFACE. It is technically called IMPLEMENTATION OF (Unitname). "Unitname" is defined in the INTERFACE file. The IMPLEMENTATION must $INCLUDE the INTERFACE it is defining at the beginning, even before the IMPLEMENTATION statement itself. The IMPLEMENTATION can USE other UNITs, nesting them as deeply as desired. (These terms will be made clearer later in this chapter, when we'll develop an IMPLEMENTATION OF Graphics to perform color graphics with Pascal.)

Attributes

Before we get any further in our discussion of systems, we need to add some more attributes to our vocabulary. We have already discussed and used the EXTERNAL and PUBLIC attributes. There are three more attributes that are similar, but not generally needed. READONLY is useful both for documentation and for preventing mistakes that might otherwise be hard to track down. Variables specified as READONLY are treated as constants and cannot be changed by the program. The version number of a program, or the number of lines on the screen, might be declared as READONLY since they would not normally change during the execution of the program.

The STATIC attribute is equivalent to declaring a variable as part of the main program, so that it doesn't get cleared on each call to the function or procedure in which it is defined. This is useful if some initialization is necessary only on the first call to a routine. All

EXTERNAL and PUBLIC declarations are automatically STATIC.

The PURE attribute can be given to functions to improve the efficiency and safety of the program. The PURE attribute indicates a function that doesn't change any data. A PURE function may not perform input or output operations, nor may it alter the value of any variable. It may only return a value.

PROGRAMs and EXTERNAL MODULEs

Sometimes, it is useful to assemble or compile portions of a program separately, and then link them together. This reduces the size of the source programs and the time it takes to compile them. On large programs several programmers might work on different portions of a system at the same time. As each portion is completed, it can be linked into the rest of the system.

The object files that we linked in previous chapters (like the Port Demonstration Program, "Pinpout", in chapter 5) are called MODULEs and the procedures within them were EXTERNAL to the programs that called them. (Remember: An EXTERNAL procedure or function is one that has already been assembled, or compiled, into an object file.)

Example Using a MODULE

The program "Stopwtch" operates just like an event timer. It is written in two parts. The first, "Stopwtch.pas", shown in Figure 7-2, is the main program. It contains the control for user inputs and display of the elapsed time. The second part is shown in Figure 7-3. It is a MODULE containing the routine "Printtime" which displays the date and time on the screen, and returns the number of seconds elapsed in the day to the main program.

Figure 7-2. Stopwatch program using EXTERNAL

```
{*****************************************************************
STOPWTCH.PAS     Stopwatch Program.
-----------------------------------------------------------------
        This program uses EXTERNAL routines to time an event
and then displays the elapsed time when ENTER is pressed.
-----------------------------------------------------------------

Program Stopwatch (input,output);

Var
        hr,min: integer;          { final elapsed time }
```

```
        sec1,sec2: real;        { number of seconds }
        ans: char;
{
-------------------------------------------------------------------
PRINTTIME       This EXTERNAL function displays the time.
-------------------------------------------------------------------
}
Function Printtime : real;      external;

-------------------------------------------------------------------
                    Main program starts here.
-------------------------------------------------------------------

Begin { stopwatch }

Repeat
        Write ('Press ENTER to start');
        Readln;
        sec1 := Printtime;      { get first time }
{
-------------------------------------------------------------------
                      Stopwatch is running.
-------------------------------------------------------------------
}
            Write ('Press ENTER for time');
            Readln;
            sec2 := Printtime;

-------------------------------------------------------------------
                     Calculate elapsed time.
-------------------------------------------------------------------
}
            sec2 := sec2 - sec1;        { get elapsed time }

                hr := trunc(sec2) div 3600;     { elapsed hours }
                sec2 := sec2 - 3600*hr;

                min := trunc(sec2) div 60;      { elapsed minutes }
                sec2 := sec2 - 60*min;          { elapsed seconds }

            Writeln (hr, ' hours  ',
                        min, ' minutes ',
                        sec2::2, 'seconds' );

        Write ('Run again ? ');
        Readln (ans);
```

```
Until (ans = 'N') or (ans = 'n');

End. { stopwatch }
!
!-----------------------------------------------------------------
Figure 7-2.      STOPWTCH.PAS     Stopwatch Program Using EXTERNALs
*****************************************************************!
```

Figure 7-3. Printing routine for stopwatch program

```
!*****************************************************************
PRNTTIME.PAS    Printing Routine for Stopwatch Program.
-----------------------------------------------------------------
       This is an EXTERNAL module that displays the date and
time, returning the time of day in seconds.
-----------------------------------------------------------------
!
Module Printtime;                ' name if initialization needed
!
!-----------------------------------------------------------------
     This routine calls these additional EXTERNAL routines.
-----------------------------------------------------------------
!
Procedure Time (var s:string);  extern; { returns time as 'HH;MM;SS'}
Procedure Date (var s:string);  extern;
Function Tics : integer;        extern;
!
-----------------------------------------------------------------
            Declare the Printtime function itself.
-----------------------------------------------------------------
!
Function Printtime : real [public];

Var
       t,d: string (8);         { time and date strings }
       sec: real;               { number of seconds during day }
       tmp: integer;            { temporary for decoding value }
       lstr: lstring (2);       { temporary for decoding dates }

Begin { Printtime }

       Time (t); Date (d);              { get date and time }
       sec := float (Tics);             { hundredths of a second }

       Writeln (d, '  ', t, '.', round(sec):2 ).
```

```
{-------------------------------------------------------------}
                Determine number of seconds in day.
{-------------------------------------------------------------}

        sec := sec / 100.0;

        Copylst (t[7], lstr);
        Concat (lstr, t[8]);
        Eval (Decode (lstr, tmp) );      { get number of seconds }
        sec := sec + float (tmp);

        Copylst (t[4], lstr);
        Concat (lstr, t[5]);
        Eval (Decode (lstr, tmp) );      { get number of minutes }
        sec := sec + float (60 * tmp);

        copylst( t[1], lstr );
        concat( lstr, t[2] );
        eval ( decode ( lstr, tmp ) );   { get number of hours }
        sec := sec + 60.0 * float( 60 * tmp );

        Printtime := sec;          { return time in seconds }

end; { print time }

End. { module }

{-------------------------------------------------------------}
```

Figure 7-3. PRNTTIME.PAS Printing Routine for Stopwatch Program

Linking a PROGRAM with Its MODULES

Linking is the combining of separately assembled or compiled modules into a program ready to be executed. The EXTERNAL attribute following a function or procedure declaration indicates that the actual routine is in a different MODULE than the calling program. The PUBLIC attribute indicates that the MODULE can be used by other programs. All functions or procedures declared in the main program as EXTERNAL must have a corresponding object file that has been declared PUBLIC.

After each program and module is compiled separately, they are then combined into a single executable program. This is done with the LINK program by specifying the object file names (PROGRAM name first) separated by a space or a plus sign (+).

When you specify the names of MODULEs to the linker, it attempts

to match up all the EXTERNALs to the PUBLICs. After loading all the MODULEs, if there are any EXTERNALs that have not yet been resolved, the linker will display an error message.

Here's how the linker would be used for the STOPWTCH example:

```
IBM Personal Computer Linker
Version 1.1Ø (C)Copyright IBM CORP 1982
Object Modules: [.OBJ] STOPWTCH+PRNTTIME
Run File [STOPWTCH.EXE]:
List File [STOPWTCH.MAP]:
Libraries| |: B.
```

Note that since the PROGRAM and MODULE must be compiled separately, we'll need to change the batch files that we've used until now for automatically compiling and linking a program. In particular, the module PRNTTIME.OBJ does not get linked into an executable program by itself.

You may find it convenient to make separate batch files for compiling and linking. A listing of these batch files is shown in Figures 7-4 and 7-5. The listings are based on the assumption that the source files are on drive A and the Pascal Diskette is on drive B.

Figure 7-4. Compile only batch file

```
*****************************************************************

time
b:pas1 %1.pas %1.obj %1.lst nul.cod
b:pas2
time

-----------------------------------------------------------------
Figure 7-4.      COMPONLY.BAT     Compile Only Batch File
*****************************************************************
```

Figure 7-5. Link only batch file

```
*****************************************************************

time
b:link %1 %2 %3 %4,%1.exe,nul,b:pascal.lib
time

-----------------------------------------------------------------
Figure 7-5.      LINKONLY.BAT     Link Only Batch File
*****************************************************************
```

The LINK program will create an executable file that can be run by typing the command "STOPWTCH". You can find the elapsed time by pressing Enter. If you tell the linker program to save the map file, PRINTTIME, DATE, TIME, and TICS will all appear in the listing of EXTERNALs. PRINTTIME is defined in the module PRNTTIME.OBJ; the rest are found in the library PASCAL.LIB.

UNIT

One of the problems that comes up when using MODULEs is that the PUBLIC and EXTERNAL declarations are in separate files. This means that if you want to change the operation of the program, you will have to make the corresponding change in every other module that may reference the one you want to change. The UNIT in IBM Pascal allows you to circumvent this problem. It provides a single place, the INTERFACE file, for declaring variables and procedures that are jointly used in several modules.

The INTERFACE

INTERFACE is one of two files associated with a UNIT (the other is the IMPLEMENTATION). The INTERFACE is not a division, and can not be compiled by itself. It must be referenced using the $INCLUDE metacommand. The INTERFACE declares the variables, functions and procedures that are to be used by several programs and implementations.

The INTERFACE of a UNIT is the portion that is used by every PROGRAM, MODULE or IMPLEMENTATION referring to the UNIT. It declares any types that are needed and specifies the types of all the variables in it. It specifies the names and parameter lists for all the procedures and functions. It does not specify EXTERNAL or PUBLIC since this is determined by whether it is to be used in an IMPLEMENTATION (PUBLIC) or in a PROGRAM or MODULE (EXTERNAL). There is no body (BEGIN...END block) declared for either the INTERFACE or any of the functions or procedures within it. The body will be provided by the PROGRAM or IMPLEMENTATION that USES the INTERFACE.

When an INTERFACE is to be used, it must be read in from the disk during the compilation. This is done by placing the $INCLUDE metacommand at the beginning of the PROGRAM or IMPLEMENTATION division. The command for including a file named "SAMPLE.INF" would be:

```
{$include:   'SAMPLE.INF'}
```

(For more information on including files, see the chapter on metacommands in the *IBM Personal Computer Pascal Compiler* manual.)

How to USE a UNIT

After all the INTERFACEs have been included, the PROGRAM or MODULE statement should appear followed by a USES statement. The names of the UNITs should appear in the USES statement. For example, assume that we want to use UNITs named SUBONE and SUBTWO in a program named FOOBAR. The INTERFACEs for each of the MODULEs are in the files SUBONE.INF and SUBTWO.INF. Then the source file for FOOBAR should start with:

```
{$include:  'SUBONE.INF'}
{$include:  'SUBTWO.INF'}

Program Foobar (input,output);

Uses subone, subtwo;            { may also be done with
                                  two USES statements }

Const
*    *    *
```

The program must start with the INTERFACE and UNIT statements. The INTERFACE file contains the actual declarations of types, constants and variables to be used. It also contains the parameter list for any procedures and functions in the UNIT. The program will not redeclare this information. Identifiers specified in the INTERFACE will not be available for use by the program unless they appear in the UNIT statement. An INTERFACE always ends with END and a *semicolon*, not a period.

The UNIT statement contains the name of the UNIT being defined and then, in parentheses, the names of all the identifiers that are to be used by other programs. The identifiers *must* include all the procedures, variables, and functions that are to be declared as EXTERNAL. Also any CONSTants or TYPEs needed should be specified, including the values of any enumerated types. Then follow the appropriate declarations for all the procedures, variables, and functions listed in the UNIT statement.

INTERFACE of a UNIT

Let's look now at a UNIT that is supplied by IBM on one of the Pascal disks. The record declaration for the "File Control Block" (FCB) is integral to the operations of Pascal files. The INTERFACE file is called "FILKQQ.INC." This file is discussed in the chapter on Files in the *IBM*

Personal Computer Pascal Compiler manual, and described in technical detail in the manual's appendix titled *File System Internals*. Here we shall describe one way to use this INTERFACE file.

First, let's review some of the fundamentals about IBM Pascal files. A file is treated as a special type of variable. Files have their own set of procedures including READLN, GET, and so on. The file can be declared either as TEXT (for ASCII use) or FILE OF ... (for binary use). There is a File Control Block associated with each file declared by a program.

The File Control Block contains all of the information needed by the system to use a file. Because of the wide range of uses of files and because of complexities involved in using them with PC-DOS, the File Control Block contains a lot of information. In Pascal this information is arranged in a RECORD type variable with many fields. We used one of these fields, MODE, when we wanted to directly access a file. For instance, to access a file named "binfile" in direct mode, we would use the assignment statement

```
binfile.mode := direct;
```

before any RESET or REWRITE statement. This statement is nothing more than an assignment to one of the fields in the File Control Block record for the file "binfile." Figure 7-6 shows the entire File Control Block INTERFACE.

Figure 7-6. File control block interface

```
(*****************************************************************)
{  IBM Personal Computer Pascal file control block  }
{    Version 1.ØØ (C) Copyright 1981 by IBM Corp    }

INTERFACE;  UNIT
  FILKQQ (FCBFQQ, FILEMODES, SEQUENTIAL, TERMINAL, DIRECT,
          DEVICETYPE, CONSOLE, LDEVICE, DISK,
          DOSEXT, DOSFCB, FNLUQQ, SCTRLNTH);

CONST

  FNLUQQ = 21;            { length of a DOS file name }
  SCTRLNTH = 512;         { length of a disk sector }

TYPE

DOSEXT = RECORD           { DOS file control block extension }
```

```
    PS  [Ø]: BYTE;        { boundary byte, not in extension }
    FG  [1]: BYTE;        { flag;  must be 255 in extension }
    XZ  [2]: ARRAY [Ø..4] OF BYTE;      { padding, internal use }
    AB  [7]: BYTE;        { attribute bits }

  END;

DOSFCB = RECORD           { DOS file control block (normal) }

    DR  [Ø]: BYTE;        { drive number, Ø=def, 1=A etc }
    FN  [1]: STRING (8);        { file name - 8 bytes }
    FT  [9]: STRING (3);        { file extension - 3 bytes }
    EX [12]: BYTE;        { current extent, lo byte }
    E2 [13]: BYTE;        { current extent, hi byte }
    S2 [14]: BYTE;        { sector size, lo byte }
    RC [15]: BYTE;        { sector size, hi byte (ext sect) }
    Z1 [16]: WORD;        { file size, lo word;  readonly }
    Z2 [18]: WORD;        { file size, hi word;  readonly }
    DA [2Ø]: WORD;        { date, bits DDDDDMMMMYYYYYYY }
    DN [16]: ARRAY [Ø..9] OF BYTE;      { reserved for DOS }
    CR [32]: BYTE;        { current sector within extent }
    RN [33]: WORD;        { direct sector number lo word }
    R2 [35]: BYTE;        { direct sector number hi byte }
    R3 [36]: BYTE;        { hi byte (if sect size < 64) }
    PD [37]: BYTE;        { pad to a word boundary, not DOS }

  END;

DEVICETYPE = (CONSOLE, LDEVICE, DISK);
               { physical device type }

FILEMODES = (SEQUENTIAL, TERMINAL, DIRECT);
               { access mode for file }

FCBFQQ = RECORD     {byte offsets start every field comment}

{fields accessible by Pascal user as <file variable>.<field>}

TRAP: BOOLEAN;      {ØØ Pascal user trapping errors if true}
ERRS: WRD(Ø)..15;   {Ø1 error status, set only by all units}
MODE: FILEMODES;    {Ø2 user file mode;  not used in unit U}

{fields shared by units F, V, U;  ERRC is write-only}

MISC: BYTE;         {Ø3 pad to word bound, unused, reserved}
ERRC: WORD;         {Ø4 error code, error exists if nonzero}
                       {1ØØØ..1Ø99:  set for unit U errors}
                       {11ØØ..1199:  set for unit F errors}
                       {12ØØ..1299:  set for unit V errors}
ESTS: WORD;         {Ø6 unused - reserved }
```

```
    CMOD: FILEMODES;     {Ø8 system file mode;  copied from MODE}

    {fields set / used by units F and V, and read-only in unit U}

    TXTF: BOOLEAN;       {Ø9 true: formatted / ASCII / TEXT file}
                             {false: not formatted / binary file}
    SIZE: WORD;          {1Ø record size set when file is opened}
                             {DIRECT: always fixed record length}
                             {others: max length (UPPER (BUFFA))}
    MISB: WORD;          {12 unused - reserved }
    OLDF: BOOLEAN;       {14 true: must exist before open; RESET}
                             {false: can create on open; REWRITE}
    INPT: BOOLEAN;       {15 true: user is now reading from file}
                             {false: user is now writing to file}
    RECL: WORD;          {16 DIRECT record number, lo order word}
    RECH: WORD;          {18 DIRECT record number, hi order word}
    USED: WORD;          {2Ø number bytes used in current record}

    {field used internally by units F and V not needed by unit U}

    LINK: ADR OF FCBFQQ;{22 DS offset address of next open file}

    {fields used internally by unit F not needed by units V or U}

    BADR: ADRMEM;        {24 ADR of buffer variable (end of FCB)}
    TMPF: BOOLEAN;       {26 true if temp file;  delete on CLOSE}
    FULL: BOOLEAN;       {27 buffer lazy evaluation status, TEXT}
    MISA: BYTE;          {28 unused - reserved }
    OPEN: BOOLEAN;       {29 file opened; RESET / REWRITE called}

    {fields used internally by unit V not needed by units F or U}

    FUNT: INTEGER;       {3Ø Unit V's unit number always above Ø}
    ENDF: BOOLEAN;       {32 Unit V's file-at-end operation flag}

    {fields set / used by unit U, and read-only in units F and V}

    REDY: BOOLEAN;       {33 reserved }
    BCNT: WORD;          {34 number of data bytes actually moved}
    EORF: BOOLEAN;       {36 true if end of record read, written}
    EOFF: BOOLEAN;       {37 end of file flag set after EOF read}

        {unit U (operating system) information starts here}

    NAME: LSTRING (FNLUQQ);   38 DOS file name for this file }
    DEVT: DEVICETYPE;         6Ø type of device accessed by this file }
    RDFC: BYTE.               61 function code to read from a device }
    WRFC: BYTE.               62 function code to write to a device }
    CHNG: BOOLEAN.            63 true if data in sbuf was changed }
    SPTR: WORD.               64 pointer (index) into sbuf }
```

```
DOSX: DOSEXT;              { 66 extended DOS file control block }
DOSF: DOSFCB;              { 74 normal DOS file control block }
IEOF: BOOLEAN;             {112 true if eoff should be true next get }
FNER: BOOLEAN;             {113 true if a filename error has occurred }
SBFL: BYTE;                {114 max length of textfile line in sbuf }
SBFC: BYTE;                {115 number of chars read into sbuf }
SBUF: ARRAY [WRD(Ø)..SCTRLNTH-1] OF BYTE;       {116 sector buffer }
PMET: ARRAY [Ø..5] OF BYTE;      {116 + sctrlnth: reserved pad }

BUFF: CHAR;                { Pascal buffer variable, (component) }

     {end of section for unit U specific OS information}

END;
END;
```

Figure 7-6. FILKQQ.INC File Control Block Interface

****

The file FILKQQ.INC can be displayed with the TYPE command in PC-DOS or examined with any word processor or editor. (Be careful not to change the contents of the file!) The file contains the INTERFACE for a UNIT named FILKQQ. After a short comment for identification, it starts with an INTERFACE statement and a UNIT statement, and ends with the END; statement. After the UNIT's name, FILKQQ, is a list of the types and constants that are used in the File Control Block. IBM provided this INTERFACE to allow the Pascal programmer to access the file system with full flexibility. Because this UNIT just contains definitions of types and constants used internally, there is no corresponding IMPLEMENTATION for the UNIT FILKQQ.

Many of the definitions in the INTERFACE are too technical for most of us, but a few of the declarations can be useful to anyone. Skip over the first couple of declarations until you find the statement:

```
FILEMODES = ( SEQUENTIAL, TERMINAL, DIRECT );
```

This defines the enumerated type FILEMODES for specifying how a file may be accessed. Note that the UNIT statement at the beginning of the INTERFACE specified both the type identifier FILEMODES, and all of the possible values: SEQUENTIAL, TERMINAL, and DIRECT. As you probably guessed this is the TYPE declaration for the MODE field in the File Control Block.

After the enumerated type definitions, we find the declaration for the File Control Block record type, FCBFQQ. This starts with:

```
FCBFQQ = RECORD
     TRAP: BOOLEAN;
     ERRS: WRD(Ø)..15;
     MODE: FILEMODES;
```

TRAP, ERRS and MODE are the three fields in the record most commonly used by Pascal programmers. TRAP is set true when the programmer wants to prevent a program from terminating when a file error is detected. ERRS is set by the DOS to indicate which error occurred, if any, when TRAP is true. We will use TRAP and ERRS again in the "Filechk" program in Figure 7-7.

If we skip down the record definition further we come to two more fields that might be useful: NAME and DEVT. NAME is the PC-DOS file name associated with the file. This is especially useful with temporary files whose names are determined by the operating system. DEVT is an enumerated value of type DEVICETYPE and it indicates what type of physical device is associated with the file. For all disk files DEVT = DISK. Non-disk files include physical devices such as the keyboard and display screen which have DEVT = CONSOLE.

IBM also supplies an INTERFACE file called FILUQQ.INC (we have been using FILKQQ.INC). FILUQQ.INC is for accessing the DOS level control of the file system and is beyond the intended scope of this book. IBM provided it for those knowledgeable assembly language programmers who want to expand the power of IBM Pascal.

PROGRAM Using a UNIT

The program "Filechk.pas", shown in Figure 7-7, USES the INTERFACE for the UNIT FILKQQ. Declared within the program is a function, "fileexists," which determines if a file already exists on the disk. This can be difficult since RESET normally will terminate the program if the file doesn't exist, and REWRITE will create a new file.

Figure 7-7. Check for file existence

```
{*****************************************************************
FILECHK.PAS     See if a file exists.
-------------------------------------,---------------------------
       This program traps DOS errors so that execution is not
terminated when an error occurs.  Instead, the F.ERRS field of the
file control block is tested to see if the file exists.
-----------------------------------------------------------------}

{$include:'FILKQQ.INC'}          { <---- Interface file          }
```

```
Program Filecheck (input, output);

Uses filkqq;                    { <---- Field Definition Unit    }

Var
        testfile: text;         { file to check }
{
-------------------------------------------------------------------
        This function does the work.  It sets the error trap and
then tries to open the file with a RESET.  Then the condition of
F.ERRS being equal to zero is assigned to the function.
-------------------------------------------------------------------
}
Function Fileexists (var f: fcbfqq) : boolean;

Begin
        f.trap := true;         { don't stop on an error        }
        Reset (f);              { try to open for input         }
        Fileexists := ( f.errs = Ø );
                                { evaluate the error flag as a
                                  Boolean value, only true if
                                  file was found                }
        Close (f);              { always close file             }
        f.trap := false;        { don't trap other errs         }
end; { fileexists }
{
-------------------------------------------------------------------
        Main program starts here.  Input a file name and check
to see if it exists.
-------------------------------------------------------------------
}
Begin { filecheck }

        Write ('Enter any file name: ');
        Readfn (testfile);      { get file name                 }

        If  Fileexists (testfile)
                Then    Writeln ('File found' )
                Else    Writeln ('File is not on the disk' );

End. { filecheck }
{
-------------------------------------------------------------------
Figure 7-7.     FILECHK.PAS     Check for File Existence
*******************************************************************
```

The only portions of FILKQQ that are used are the File Control Block record type, FCBFQQ, and two of the fields within the record, TRAP and ERRS. Note that the file that contains the INTERFACE, FILKQQ.INC, is specified by an $INCLUDE metacommand at the beginning of the program. This file has to be on the default disk drive or the compiler will give a fatal error. After the $INCLUDE comes the division specifier: in this case the PROGRAM statement. Immediately after the PROGRAM statement comes the USES statement specifying the name of the UNIT to be used, FILKQQ.

The main body of the program is a simple routine that associates a file name entered by the user with a file type variable "testfile". A call is then made to the function "fileexists" and an appropriate message is displayed based upon the results of the function call. Note that the variable "testfile" could also have been declared as a binary type file (FILE OF...) without changing the results.

The function "fileexists" utilizes the INTERFACE file. It takes a single parameter, "f" of type FCBFQQ. The error trapping is set TRUE for the file to prevent termination of the program if an attempt is made to open the file for input (the RESET statement), and the file does not exist. This condition will be signaled by the value of the File Control Block field ERRS. The file is always left CLOSEd and the error trapping flag is always returned to its default state, FALSE.

This example is informative and is a useful addition to a library of user written procedures and functions. The function "fileexists" could be put into a module by itself by changing the PROGRAM statement to a MODULE statement and deleting the declaration of "testfile" and the body of the main program. When used in its present form, the program requires no special handling by the linker. Also notice that we used a portion of a large complex record definition without having to see the entire definition; this minimized the risk of typing errors. This capability is one of the advantages of using UNITs.

The IMPLEMENTATION OF a UNIT

We have seen how to take frequently-used codes and set up a MODULE, and how, using an INTERFACE file, the main program can easily use a complicated definition. These features can be combined by using an IMPLEMENTATION. Like MODULEs, an IMPLEMENTATION is a separately-compiled source file that contains variables and procedures used by one or more other programs. The same INTERFACE file that the IMPLEMENTATION uses to define the EXTERNAL procedure can be used by the calling program(s).

An IMPLEMENTATION is similar in structure to a PROGRAM or a

MODULE. It must begin with an $INCLUDE for the INTERFACE for the UNIT that it defines. Any other required UNITs should also be $INCLUDEd at the beginning of the IMPLEMENTATION. Instead of a PROGRAM or MODULE statement, the IMPLEMENTATION OF statement is used. The USES statement must follow with the name of the UNIT. Any declarations that are needed only for local use within the IMPLEMENTATION should be made. LABELs can be used and a VALUE section specified. These are useful features not available in MODULEs.

We will use these IBM compilands in the remainder of this chapter to develop a system of programs to perform color graphics output from Pascal on an IBM PC equipped with the color graphics adapter.

The Color Graphics System

Among the many features available for the IBM Personal Computer is a fine color graphics display adapter. The adapter allows the IBM PC to completely control the operation of a color monitor or color television. This makes it possible to write programs that will output color graphics: a very desirable feature in any computer. As the proverb says, "One picture is worth a thousand words." Color graphics can be used for a number of applications, from games to sales charts, and greatly enhance the value of your PC.

The BASIC language for the IBM PC supports color graphics operations fully. There are special BASIC commands that control the display mode and color. Unfortunately, the Pascal language is lacking these features. Perhaps at some future date IBM will release an updated version of Pascal, containing some graphics capability. For now, we will develop our own color graphics system as an example of a system of programs.

The Hardware

To perform color graphics with the IBM PC, we must have a machine equipped with the color graphics adapter, shown in Figure 7-8. The color adapter is a printed circuit board that plugs into one of the expansion slots inside the IBM PC. It has 16K of its own memory which is mapped into the system memory space, and used as the display buffer. Its operations are all controlled by a CRT controller chip, located on the board. This is the same kind of controller that is used for the monochrome display. The adapter is also controlled through the CPU's INPUT/OUTPUT ports.

The adapter can be connected directly into a color monitor. It can also output a video signal to an external RF modulator for a standard color television set.

The Display Buffer

The 16K of memory on the adapter board is used as the display buffer, shown in Figure 7-9. This means that whatever is to appear on the screen is represented in the buffer in some binary format. The display buffer physically resides on the color adapter board and is mapped into the system memory space starting at address B8000h ("h" indicates a hexadecimal value). Like the monochrome display buffer, this memory is accessible by both the CRT controller and the CPU.

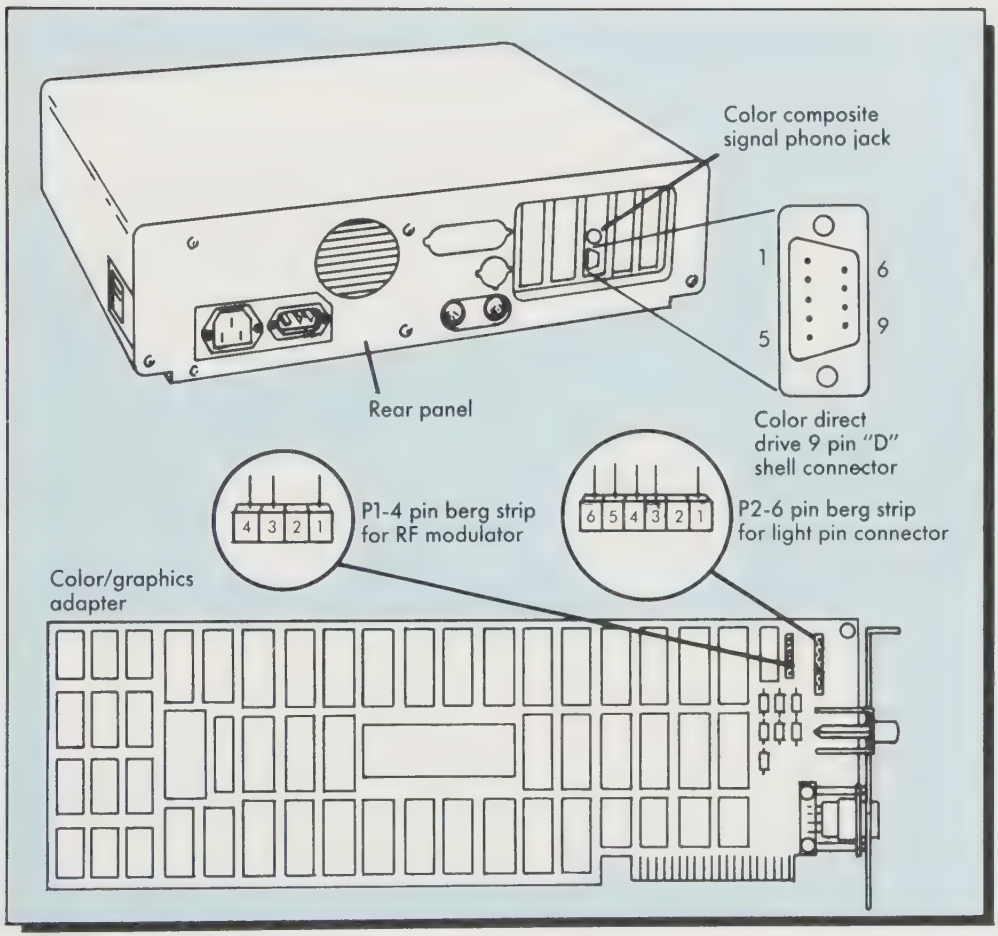


Figure 7-8. Color graphics adapter hardware

Like the monochrome display buffer, the color display buffer is a contiguous array of memory locations. These are used and interpreted differently, depending upon the current mode of the adapter.

Color Graphics Modes

There are two basic modes in which the color graphics adapter may be operated.

ALPHANUMERIC	supports the full 256 ASCII characters with standard attributes.
ALL POINTS ADDRESSABLE	graphics mode has 2 levels of resolution. The medium resolution, 320x200, supports color graphics; the high resolution, 640x200, supports black and white only.

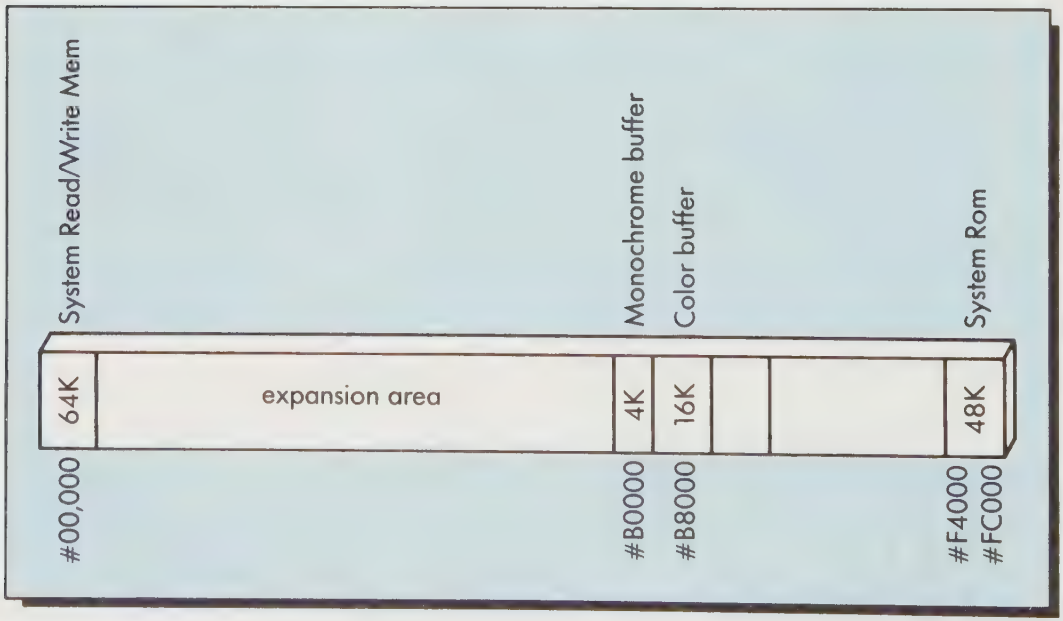


Figure 7-9. Color display buffer in system memory

Alphanumeric Mode

The alphanumeric mode is used to display the standard character set on the color adapter. This allows the user to have a color television as the only display device on the system. In this mode, the characters are displayed much the same as on the monochrome display. The display buffer is used to contain the characters to be displayed, along with their attributes. Two bytes are used for each character in the display buffer. (See Figure 7-10.)

The first byte contains the ASCII code for the character. This code is used to access a table in which the actual dot pattern for the character is stored. This table can be at any location in the system memory since its address can be programmed into the CRT controller. This means that you can build your own set of characters somewhere in memory, and access it by changing the table address for the controller. Each character is displayed as an 8x8 matrix. Every consecutive 64 bits (8 bytes) in the character generator table contain the pattern for a single character.

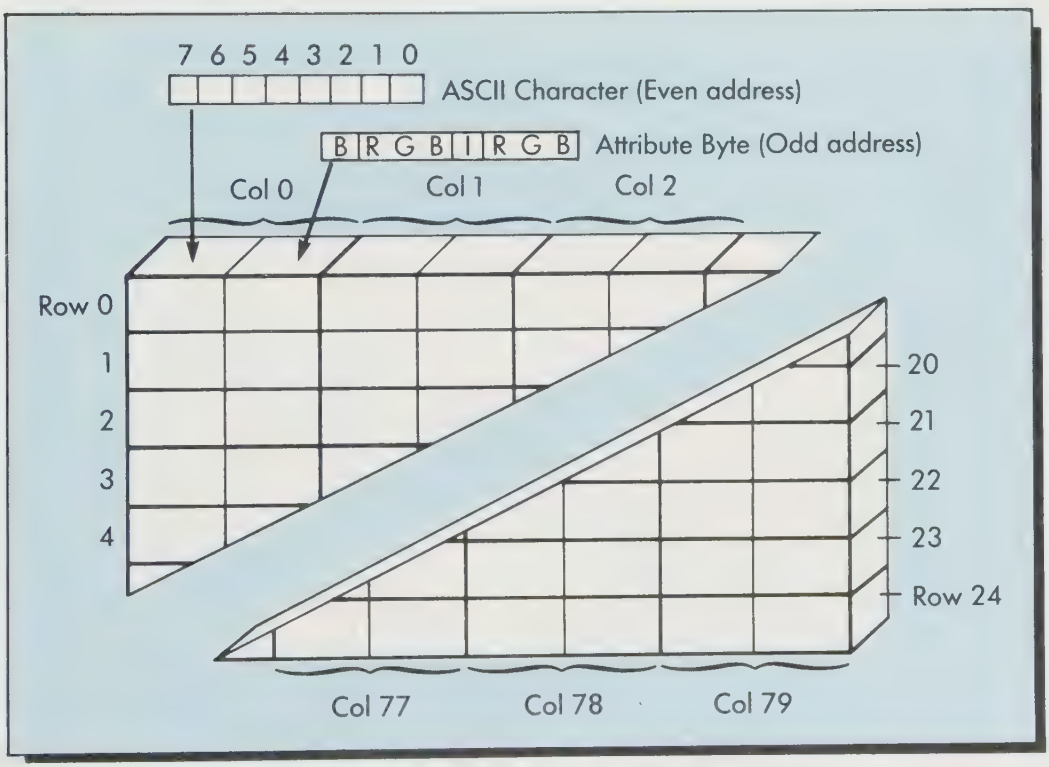


Figure 7-10. Buffer in alphanumeric mode

The second byte for each character in the buffer contains the attribute information. These attributes are similar to the attributes given to characters in the monochrome display. In addition to the *intensity* and *blink* attributes, here we can specify one of eight colors. If you intend to pursue this area further, we suggest that you have the *IBM Personal Computer Technical Reference* manual available.

All Points Addressable Modes

As we mentioned earlier, the all points addressable graphics mode has two levels of resolution: medium and high.

Medium Resolution Mode. Figure 7-11 shows the medium resolution color mode. It will probably be your most common mode for general color graphics. The screen is divided into 200 horizontal lines, each containing 320 picture elements, called *pels* by IBM.

Each picture element may be set to one of four colors. This is possible because there are two bits used to represent each picture element. So, there will be four picture elements stored in each byte of the display buffer. Accessing a particular picture element will require some arithmetic to determine which byte actually contains the picture element.

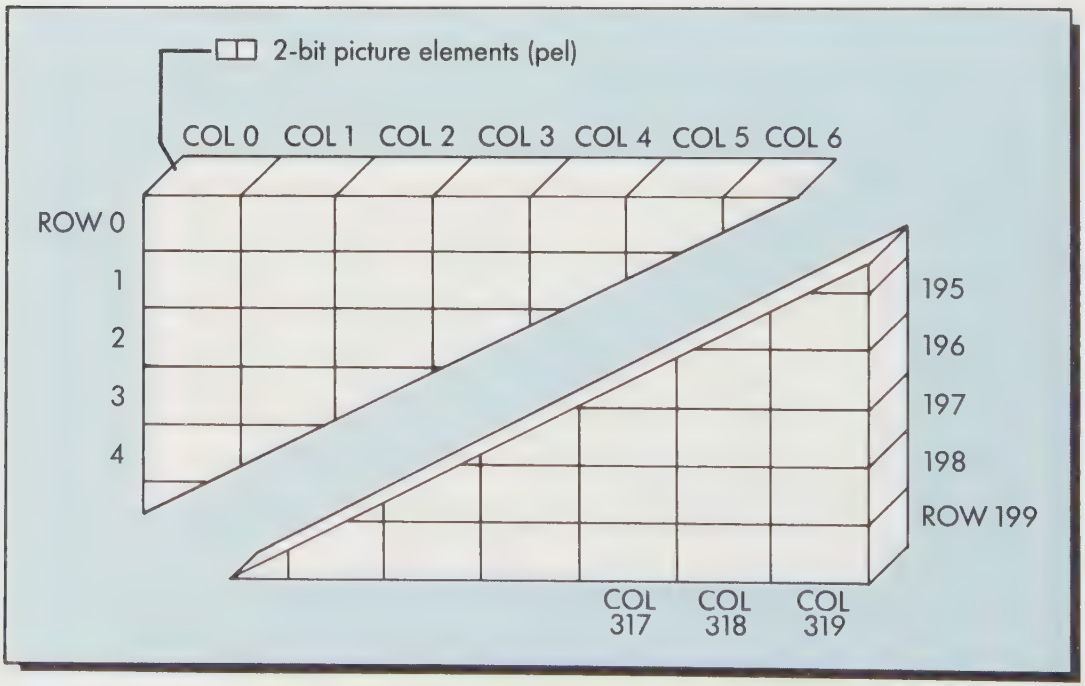


Figure 7-11. Medium resolution color mode

High Resolution Mode. The color options available are decreased in the high resolution mode. As shown in Figure 7-12, the screen is divided into 200 horizontal lines, each containing 640 picture elements.

Each bit in the display buffer represents a picture element. That's why this mode only supports black and white: the one bit can only determine whether the picture element is on or off.

Odd and Even Scan Lines. When operating in either the medium resolution or high resolution graphics mode, the display buffer is no longer treated as a contiguous array of screen locations. Instead, the odd and even scan lines are grouped together.

As illustrated in Figure 7-13, the first part of the display buffer contains the even numbered lines (0,2,4,etc.). Then, there is a small area that is unused. Finally, the odd numbered lines (1,3,5,etc.) occupy the rest of the buffer. This throws a wrinkle into our color graphics system since we can't treat the color display buffer quite so simply as we treat the monochrome buffer.

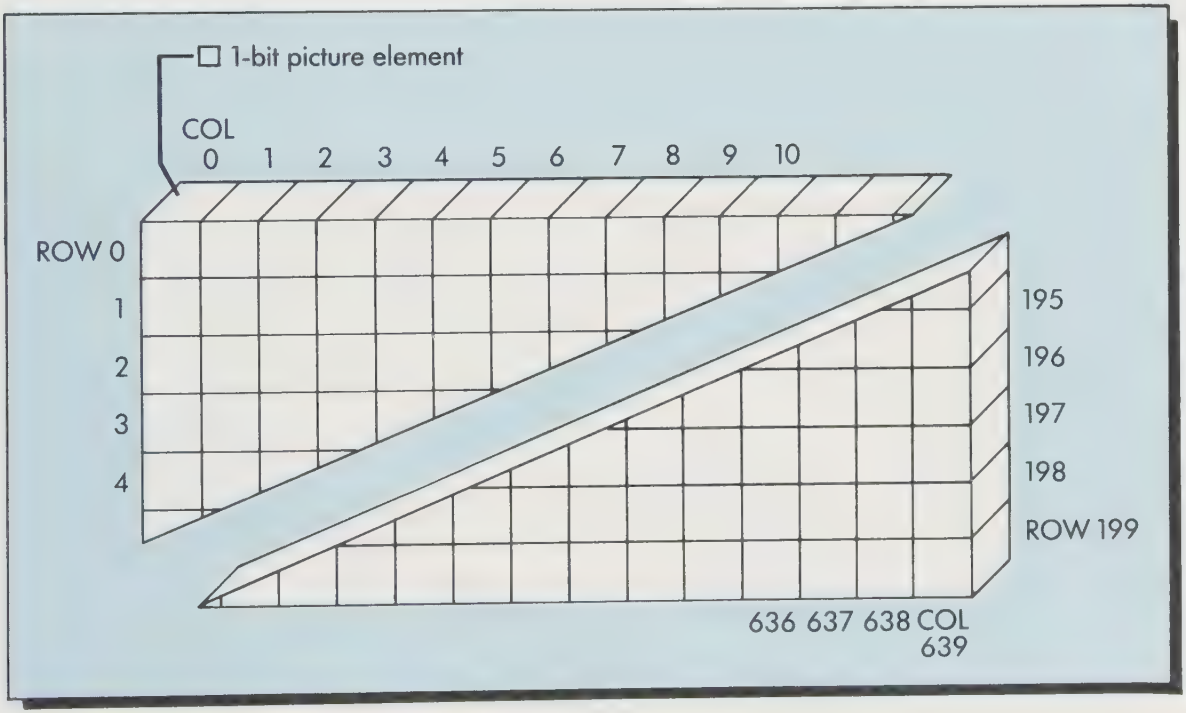


Figure 7-12. High resolution black and white mode

Direct Pascal Access

We could access the color display buffer in much the same way as we accessed the monochrome display buffer in chapter 5. By using an ADDRESS type variable. we can access any memory location, including those in the display buffer.

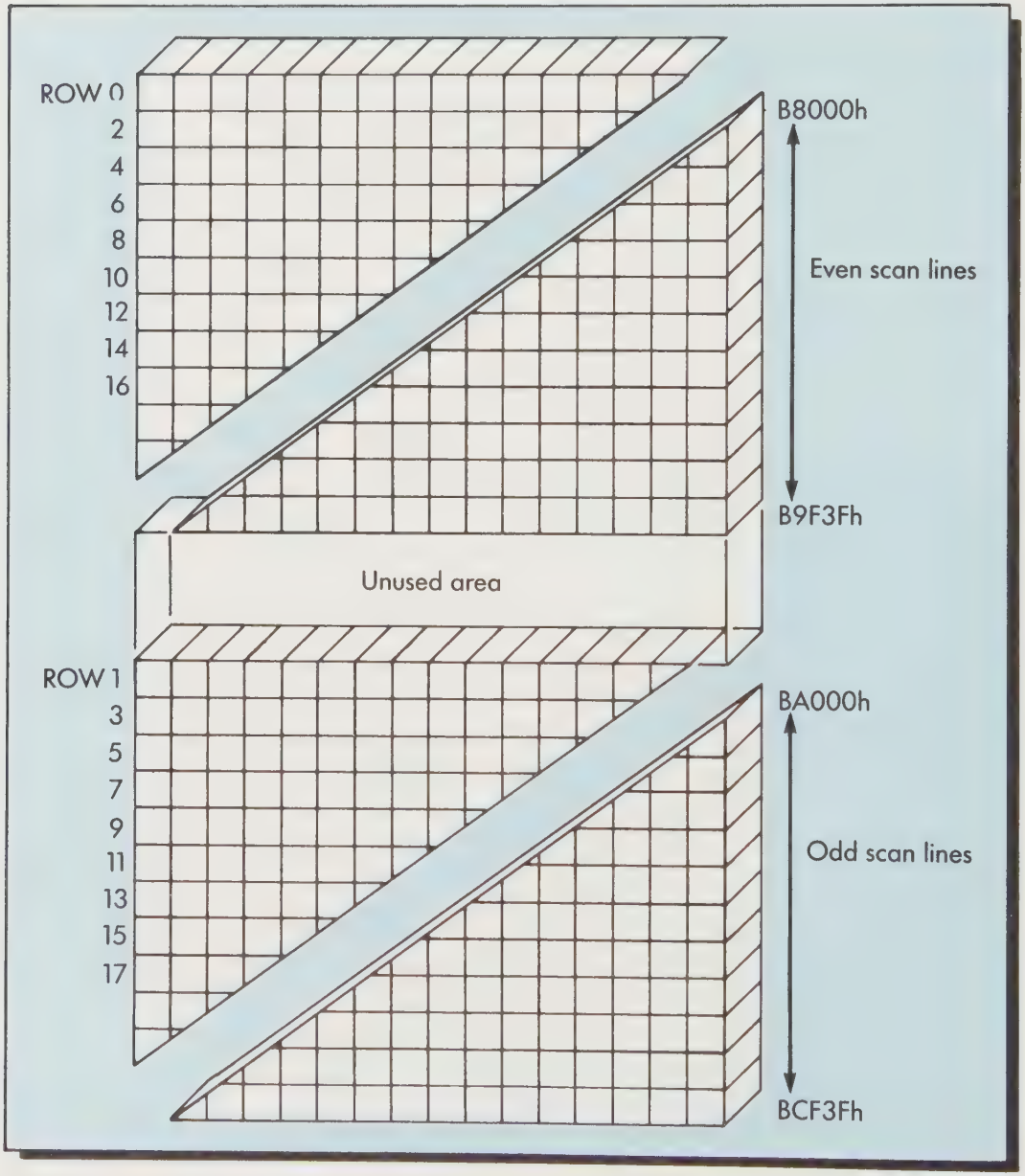


Figure 7-13. Even and odd scan lines

When we were developing the program examples for this book, we tried this approach for a while, and decided that it was not acceptable, for two reasons. First, the declaration for the display buffer with its segregated odd and even scan lines and its different modes, makes for a complicated Pascal program. Second, the complexity of addressing the display buffer slowed the execution speed until it was not satisfactory for most applications. For these reasons, we decided to use some assembly language routines to perform the primitive functions for our graphics system.

Assembly Routines for Color Graphics

To use our assembly routines for color graphics you must have the IBM Assembler or a similar program. Several operations will be standard in our system. These will be performed by the assembly language routines set forth in Figures 7-14 through 7-18. These routines are written in assembly language, as PUBLIC procedures. Once assembled, they will reside on the disk as object files which can be linked into any main program.

We are presenting the assembly source statements for these routines so that you can see some basic things about their use. If you would like to learn more about assembly language, we suggest that you also read *Assembly Language Primer for the IBM PC and XT* by Robert Lafore (New York: Plume/Waite, New American Library, 1984), another book in this series.

The routine BANDW, like the rest of our graphics routines, utilizes the DOS to perform the actual I/O to the color adapter. The three mode setting routines are examples of this technique. The proper code is placed in the "equipment byte" and a call to the DOS is made via an interrupt instruction (INT). The routines COLORM and HIGHRES, set forth in Figures 7-15 and 7-16, are similar to BANDW.

To facilitate the actual graphics operations, we have developed two additional assembly language routines, PLOT and DRAW. These are set forth in Figures 7-17 and 7-18.

PLOT is a procedure that treats the high resolution screen like a giant chessboard. Each picture element can be described by its row and column position, expressed as X and Y coordinates. The range along the X axis is from 0 to 639, while the range along the Y axis is from 0 to 199. The third parameter to the PLOT routine is the color. In high resolution mode, there are only two colors: black and white. A white point will be plotted on the screen if the value of color is 1. A black point will be plotted (blanking out any point already existing) if the value of color is 0.

The DRAW routine is used for drawing lines on the screen. Each line is defined with two pairs of X and Y coordinates and a color value. Each coordinate pair describes one end of the line.

The routine will operate in either the high resolution (black and white) mode, or the medium resolution (color) mode. The only differences are that in the medium resolution mode, the X axis ranges from 0 to 319, and the color parameter can be set to 0 (for black) or 1, 2, or 3 for one of the colors in the medium resolution palette.

Figure 7-14. Set monochrome display mode

```
;*************************************************************
;BANDW.ASM      Set Monochrome Display Mode.
;-------------------------------------------------------------
;       This routine sets the system monochrome display mode by
;encoding its value into the "equipment flag" and making a BIOS
.call.
;-------------------------------------------------------------
;SEGMENT TO CONTAIN EQUIPMENT FLAG

rom_da  segment at 40h
        org 10h
eq_flag label  word
rom_da  ends

;-------------------------------------------------------------
;PROGRAM SEGMENT

coder   segment para    public
        assume cs:coder
public  bandw

;-------------------------------------------------------------
bandw   proc far
        push  ds        ;save DS

;CHANGE TO MONOCHROME DISPLAY

        assume ds:rom_da
        mov  ax,rom_da  ;set DS to
        mov  ds,ax      ; equipment flag

        mov  ax,eq_flag   ;get equipment flag
        and  ax,11001111b ;mask off video bits
        or   ax,00110000b ;monochrome bits
        mov  eq_flag,ax   ;back into flag
```

```
        mov  al,2       ;8Ø column b & w code
        mov  ah,Ø       ;"set mode" function
        int  1Øh        ;call Video BIOS

        pop  ds         ;get DS back

        ret             ;return to caller

bandw   endp
coder   ends
        end

;-----------------------------------------------------------------
;Figure 7-14.    BANDW.ASM       Set Monochrome Display Mode
;*****************************************************************
```

Figure 7-15. Set medium resolution color mode

```
;*****************************************************************
;COLORM.ASM     Set 32Ø x 2ØØ Medium Resolution Color Mode
;-----------------------------------------------------------------
;       This routine sets the medium resolution color mode by
;encoding its value into the "equipment flag" and making a BIOS
;call.
;-----------------------------------------------------------------
;SEGMENT TO CONTAIN EQUIPMENT FLAG

rom_da  segment at 4Øh
        org 1Øh
eq_flag label  word
rom_da  ends

;-----------------------------------------------------------------
;PROGRAM SEGMENT

coder   segment para    public
        assume cs:coder
public  colorm

;-----------------------------------------------------------------
colorm  proc far
        push  ds        ;save DS

;CHANGE TO 32Ø x 2ØØ color mode

        assume ds:rom_da
        mov  ax,rom_da  ;set DS to
        mov  ds,ax      ; equipment flag
```

```
        mov  ax,eq_flag   ;get equipment flag
        and  ax,11Ø01111b ;mask off video bits
        or   ax,ØØ1ØØØØØb ;color card 8Ø x 25
        mov  eq_flag,ax   ;back into flag

        mov  al,4         ;32Øx2ØØ color
        mov  ah,Ø         ;"set mode" function
        int  1Øh          ;call Video BIOS

        pop  ds           ;get DS back

        ret               ;return to caller

colorm  endp
coder   ends
        end


;-----------------------------------------------------------------
;Figure 7-15.   COLORM.ASM      Set Medium Resolution Color Mode
;*****************************************************************
```

Figure 7-16. Set high resolution black and white mode

```
;*****************************************************************
;HIGHRES.ASM    Set 64Ø x 2ØØ High Resolution Graphics Mode.
;-----------------------------------------------------------------
;       This routine sets the high resolution color graphics mode
;by placing its value into the "equipment flag" and making a BIOS
;call.
;-----------------------------------------------------------------
;SEGMENT TO CONTAIN EQUIPMENT FLAG

rom_da  segment at 4Øh
        org 1Øh
eq_flag label  word
rom_da  ends

;-----------------------------------------------------------------
;PROGRAM SEGMENT

coder   segment para    public
        assume cs:coder

;-----------------------------------------------------------------
highres proc far
        push  ds        ;save DS

;SET UP FOR 64Ø x 2ØØ black and white
```

```
        assume ds:rom_da
        mov  ax,rom_da  ;set DS to
        mov  ds,ax      ; equipment flag

        mov  ax,eq_flag   ;get equipment flag
        and  ax,11001111b ;mask off video bits
        or   ax,00100000b ;color card 80 x 25
        mov  eq_flag,ax   ;back into flag

        mov  al,6       ;640x200 b & w code
        mov  ah,0       ;"set mode" function
        int  10h        ;call Video BIOS

        pop  ds         ;get DS back

        ret             ;return to caller

highres endp
coder   ends
        end
```

```
;-----------------------------------------------------
;Figure 7-16.   HIGHRES.ASM     Set High Resolution Black and
;                               White Mode
;*****************************************************
```

Figure 7-17. High resolution plot routine

```
;*****************************************************
;PLOT.ASM        Color Graphics Plot Routine
;-----------------------------------------------------
;       This routine uses the BIOS to plot a point in the
;color graphics 640 x 200 high resolution mode.
;-----------------------------------------------------

pro_nam segment para    public
        assume  cs:pro_nam
public  plot
plot    proc    far     ;main part of program

;-----------------------------------------------------
;Get the parameters from the stack.  Assume stack looks like this
;-----------------------------------------------------
    0   <-------stack pointer
    1
    2   hi-offset return address
    3   lo
    4   hi-segment return address
    5   lo
    6   hi color word
```

```
;   7   lo
;   8   hi column number
;   9   lo
;  1Ø   hi row number
;  11   lo

        push bp             ;save old BP
        mov  bp,sp          ;ptr to stack
        mov  ax,[bp + 6]    ;get color word
        mov  cx,[bp + 8]    ;column number
        mov  dx,[bp + 1Ø]   ;row number

;-----------------------------------------------------------------
;       Use the BIOS to write dot.
;               row # in DX
;               col in CX
;               color in AX
;-----------------------------------------------------------------
        mov   ah,12d    ;write dot function
        int   1Øh       ;video BIOS routine

        pop   bp        ;restore old BP
        ret             ;return to caller

plot    endp    ;end of main part of program
pro_nam ends    ;end of code segment
        end
;-----------------------------------------------------------------
;Figure 7-17.   PLOT.ASM        High Resolution Plot Routine
;*****************************************************************
```

Figure 7-18. Color graphics line routine

```
;*****************************************************************
;DRAW.ASM       Color Graphics Line Drawing Routine.
;-----------------------------------------------------------------
;       This routine uses the BIOS to perform the actual
;color graphics I/O, in the current mode.
;-----------------------------------------------------------------
datarea segment         ;define data segment

delta_x dw      ?       ; |x2-x1|
delta_y dw      ?       ; |y2-y1|
halfy   label   word    ; |y2-y1| / 2
halfx   dw      ?       ; |x2-x1| / 2
count   dw      ?       ; set to long axis

datarea ends
```

```
;-----------------------------------------------------------------
pro_nam segment para    public
        assume  cs:pro_nam,ds:datarea
        public  draw
draw    proc    far

;-----------------------------------------------------------------
;DRAW -- SUBROUTINE TO DRAW LINE
;
;Input is x1, y1  (start of line)
;         x2, y2  (end of line)
;         color (Ø-1 or Ø-4)

;Assume these five 16-bit numbers are
;on the stack in the form
;-----------------------------------------------------------------
;Ø      old BP register  <---stack pointer
;1        lo
;2      offset return address
;3        lo
;4      segment return address
;5        lo
;6      color   ;bp + 6
;7        lo
;8      y2 hi   ;bp + 8
;9        lo
;1Ø     x2 hi   ;bp + 1Ø
;11       lo
;12     y1 hi   ;bp + 12
;13       lo
;14     x1 hi   ;bp + 14
;15       lo
;-----------------------------------------------------------------
        push bp         ;save calling prog's BP
        mov  bp,sp      ;establish new BP

;find |y2-y1| -- result is delta_y
        mov  ax,[bp + 8]  ;get y2
        sub  ax,[bp + 12] ;subtract y1
                          ;result in AX
;figure out if delta_y is positive or negative
;  SI=1 if positive, SI=-1 if negative
        mov  si,1       ;set flag to positive
        jge  store_y    ;  keep it that way
        mov  si,-1      ;set flag to negative
        neg  ax         ;set to abs value
store_y:
        mov  delta_y,ax ;store delta_y
```

```
;find |x2-x1| -- result is delta_x
        mov  ax,[bp + 1Ø]  ;get x2
        sub  ax,[bp + 14]   ;subtract x1
                           ;result in AX
;figure out if delta_x is positive or negative
;  DI=Ø if positive, DI=1 if negative
        mov  di,1          ;set flag to positive
        jge  store_x       ;  keep it that way
        mov  di,-1         ;set flag to negative
        neg  ax            ;set to abs value
store_x:
        mov  delta_x,ax ;store delta_x

;figure out if slope is greater than 1, or less
        mov  ax,delta_x ;get delta_x
        cmp  ax,delta_y ;compare deltas
        jl   csteep     ;slope > 1
        call easy       ;slope < 1, or = 1
        jmp  finish

csteep:
        call steep      ;slope > 1

;DONE LINE -- RETURN
finish:
        pop  bp
        ret  1Ø         ;return to caller

draw    endp

; ------------------------------------------------------------------
easy    proc  near

;SLOPE < 1

;calculate half of delta_x, call it halfx
        mov  ax,delta_x ;get |x2-x1|
        shr  ax,1       ;shift right to divide
        mov  halfx,ax   ;  by 2

;initialize values
        mov  cx,[bp+14] ;set x1
        mov  dx,[bp+12] ;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

        ret             ;done line

easy    endp

;---------------------------------------------------------------
steep   proc  near

;SLOPE > 1

;calculate half of delta_y, call it halfy
        mov  ax,delta_y ;get |y2-y1|
        shr  ax,1       ;shift right to divide
        mov  halfy,ax   ;  by 2

;initialize values
        mov  cx,[bp+14] ;set x1
        mov  dx,[bp+12] ;set y1

        mov  bx,Ø       ;initialize BX
        mov  ax,delta_y ;set count
        mov  count,ax   ;  to x2-y1

newdot2:
        call dotplot    ;plot the dot
        add  dx,si      ;inc/dec Y
        add  bx,delta_x ;add |x2-x1| to BX
        cmp  bx,halfy   ;compare to |y2-y1| / 2
        jle  dcount2    ;don't inc/dec X
        sub  bx,delta_y ;subtract |y2-y1|
                        ;  from BX
        add  cx,di      ;inc/dec X
dcount2:
        dec  count      ;done line yet?
        jge  newdot2    ;not yet
        ret             ;return to main dline

steep   endp
```

```
;-----------------------------------------------------------------
dotplot proc near

;SAVE REGISTERS AND CALL PLOT ROUTINE

        push bx         ;save registers
        push cx
        push dx
        push ax
        push si
        push di

;use ROM routine to write dot
;requires row # in DX, col in CX, color in AL

        mov   ax,[bp + 6]  ;set color value
        mov   ah,12d       ;write dot function
        int   1Øh          ;video BIOS routine

        pop  di         ;restore registers
        pop  si
        pop  ax
        pop  dx
        pop  cx
        pop  bx

        ret             ;return

dotplot endp

;-----------------------------------------------------------------
pro_nam ends    ;end of code segment
        end     ;end assembly

;-----------------------------------------------------------------
;Figure 7-18.   DRAW.ASM        Color Graphics Line Routine
;*****************************************************************
```

IMPLEMENTATION OF Graphics

In order to use all of these new routines in Pascal programs, we have to set up an INTERFACE file, containing the declarations of all the external routines in the UNIT named "Graphics." Figure 7-19 shows the INTERFACE file.

Each of the external routines is declared, including the parameter list. These are all then grouped together under the UNIT name, "Graphics." This file must be $INCLUDEd in each compilation that is to use the graphics routines.

Figure 7-19. Graphics interface file

```
{*****************************************************************
GRAPHICS.INF    Color Graphics Interface File.
-----------------------------------------------------------------
}
Interface;

Unit graphics (bandw,colorm,highres,plot,draw,box);

Procedure bandw;                                    { b/w mode }
Procedure colorm;                                   { color mode }
Procedure highres;                                  { high res mode }
Procedure plot (x,y,color:integer);                 { plot a point }
Procedure draw (x1,y1,x2,y2,color:integer);         { draw a line }
Procedure box (x,y,xdim,ydim,color:integer);        { draw a box }

end; { graphics interface }
{
-----------------------------------------------------------------
Figure 7-19.    GRAPHICS.INF    Graphics Interface File
*****************************************************************}
```

The IMPLEMENTATION OF Graphics is like part of a program. It is compiled into an OBJect file, which can then be linked into many executable programs. Figure 7-20 is a listing of the IMPLEMENTATION file. Notice the external declarations. Here they don't need their respective parameter lists; they have already been declared in the INTERFACE.

Figure 7-20. Graphics implementation file

```
{*****************************************************************
GRIMPLNT.PAS    Color Graphics Implementation.
-----------------------------------------------------------------
}

{$include:'graphics.inf'}       { <---- Interface file          }

Implementation of Graphics;
{
-----------------------------------------------------------------
       External Assembly Language Procedure Declarations.
-----------------------------------------------------------------}
Procedure bandw;        external;
Procedure colorm;       external;
Procedure highres;      external;
Procedure plot;         external;
Procedure draw;         external;
```

```
{
------------------------------------------------------------
              Pascal procedure for drawing boxes.
------------------------------------------------------------}
Procedure box;

        Var
                x1,y1,x2,y2,col :integer;

        Begin
                x1 := x;                y1 := y;
                x2 := x1 + xdim;        y2 := y1;
                draw (x1,y1,x2,y2,col);

                y2 := y1 + ydim;        x1 := x2;
                draw (x1,y1,x2,y2,col);

                x2 := x;                y1 := y2;
                draw (x1,y1,x2,y2,col);

                x1 := x;                y2 := y;
                draw (x1,y1,x2,y2,col);

        end; { box }

end. { graphics implementation }
{
------------------------------------------------------------
Figure 7-2Ø.   GRIMPLNT.PAS   Color Graphics Implementation File
************************************************************}
```

One additional feature we thought would be useful is the BOX routine. This Pascal procedure uses the assembly language primitive DRAW to draw the four lines that comprise a box. The parameter list for the box routine describes the location of the upper left corner of the box, and both the X and Y dimensions. From this, the Pascal routine can determine the end points of the four lines that must be drawn to complete the box. You can use the BOX procedure as a model for your own procedures.

Pascal Demonstration Programs

To illustrate the use of these new routines, we've put together some demonstration programs. These are nothing fancy, nor are they difficult to understand. They do provide insight into the operation of the color graphics adapter, and are a convenient springboard for further experimentation.

Linking the Graphics Programs

After each one of the Pascal demonstration programs has been compiled, you will need to link it with all of the external routines before an executable file can be created. Here is what the linker session might look like.

```
IBM Personal Computer Linker
Version 1.1Ø (C)Copyright IBM CORP 1982
Object Modules: [.OBJ] PROGNAME+
                       GRIMPLNT+
                       BANDW+COLORM+HIGHRES+
                       PLOT+DRAW
Run File [PROGNAME.EXE]:
List File [PROGNAME.MAP]:
Libraries[ ]: B:
```

The PLOTDEMO Program

The first program, "Plotdemo", shown in Figure 7-21, simply plots two functions in the high resolution mode. The first is a sawtooth pattern consisting of three lines. The second function is a lazy parabola generated using multiplication rather than the SQR function. Notice the $INCLUDE of the INTERFACE file at the very beginning of the compilation. This comes even before the PROGRAM declaration. Also notice the USES statement. This refers to the UNIT named "Graphics" that has already been compiled into the IMPLEMENTATION OF Graphics, and stored on the disk as an OBJ file.

Figure 7-21. High resolution plot demonstration

```
{*****************************************************************}
PLOTDEMO.PAS    Color Graphics Plot Demo Program.
-----------------------------------------------------------------
       This program demonstrates the use of the color graphics
routines to set the display mode and plot a series of points.
The program first draws a sawtooth pattern of lines, and then
a lazy parabola using REAL arithmetic.
-----------------------------------------------------------------
}

{$include:'graphics.inf'}
```

```
Program Plotdemo (input,output);

Uses Graphics;

Var
    x,y,z       :integer;
    xx,yy       :real;
    select      :char;

{
-------------------------------------------------------------------
                        Select the mode.
-------------------------------------------------------------------
}
Begin

        bandw;                  { <---- Make sure in alpha mode }

        Writeln(' h - high res');
        Writeln(' m - medium res');
        Writeln(' l - low res');
        Write  ('Select mode ');        Read(select);

        Case select of
                'h':    highres; { <---- Set 200x640 mode       }
                'm':    colorm;  { <---- Set 200x320 mode       }
                'l':    bandw;   { <---- Set 80x25 mode         }
        end;
{
-------------------------------------------------------------------
                    Draw a sawtooth pattern.
-------------------------------------------------------------------
}
        For y := 0 to 199 Do
                [
                x := y;
                plot (y, x, 1);
                ];

        For y := 199 downto 0 Do
                [
                x := 400 - y;
                plot (y, x, 1);
                ];

        For y := 0 to 199 do
                [
                x := y + 400;
                plot (y, x, 1);
                ];
```

```
{
-------------------------------------------------------------------
                    Draw a lazy parabola.
-------------------------------------------------------------------

        For y := Ø to 199 do
                {
                yy := float(y-1ØØ) / 1ØØ;
                xx := 6ØØ * yy * yy;
                x := trunc(xx);
                plot (y, x, 1);
                };

        bandw;                  { <---- Return to monochrome    }

End.
}
-------------------------------------------------------------------
```

Figure 7-21. PLOTDEMO.PAS High Resolution Plot Demonstration

The LINDEMO1 Program

Shown in Figure 7-22 is one of two programs that demonstrate the use of the DRAW routine. In this example, you are prompted to enter the end points for the line as INTEGERs. As soon as the four values have been entered, the routine proceeds to draw the line. Notice that with each successive line, the whole screen scrolls up one line. This is because we are using standard Pascal TERMINAL mode I/O for the keyboard and screen.

Again, notice the use of INTERFACE, IMPLEMENTATION OF, and USES in this example.

Figure 7-22. Single line demonstration

```
{*****************************************************************
LINDEMO1.PAS    Color Graphics Line Drawing Demo.
-----------------------------------------------------------------
        This program demonstrates the use of the DRAW routine.
This routine operates only in the high resolution 64Ø x 2ØØ mode.
It inputs endpoint pairs from the user in the form X and Y.
-----------------------------------------------------------------
}
{$include:'graphics.inf'}       { <---- Interface file          }

Program Lindemo1 (input, output);

Uses Graphics;                  { <---- Graphics unit           }
```

```
Var
        mode            :char;

        x1, y1, x2, y2  :integer; { <---- Endpoint pairs        }

{-------------------------------------------------------------------
        Main program starts here.  The program will terminate
after drawing a line with an x1 value greater than 640.
-------------------------------------------------------------------}

Begin

        Writeln (' m - medium res 320 x 200');
        Writeln (' h - high res 640 x 200');
        Writeln ('Enter mode ------------> ');
        Readln (mode);

        Case mode of
            'm' :Colorm;
            'h' :Highres;
                Otherwise Abort ('lindemo1', #0000, #0000);
                        end;

    Repeat
        Writeln ('Enter x1, y1, x2, y2');
        Readln (x1, y1, x2, y2);

        Draw (x1, y1, x2, y2, 1); { <---- Draw the line

    Until x1 > 640;
End.

{-----------------------------------------------------------------
Figure 7-22.    LINDEMO1.PAS    Single Line Demonstration
*****************************************************************}
```

The LINDEMO2 Program

In Figure 7-23, we let the program calculate the end points of the line. All of the lines are drawn radiating outward from the center of the screen. The lines are drawn in a fan motion starting at the upper left corner of the screen. You may enter the "interval," which is simply the step used to fan the lines around the screen. You may also specify the color from one of the three colors in the currently defined palette.

Figure 7-23. Demonstrates lines from center of screen

```
{*****************************************************************
LINDEMO2.PAS    Draws Lines from the Center of the Screen.
-----------------------------------------------------------------
}

{$include:'graphics.inf'}       { <---- Interface file          }

Program Lindemo2 (input,output);

Uses Graphics;

Var
        x1,y1,x2,y2,col,
        count,xmax,ymax,interval
                                        :integer;
        answer  :char;

Begin

    Repeat

        Writeln('Enter interval ');     Read(interval);
        xmax := 32Ø div interval;
        ymax := 2ØØ div interval;

        Write ('Enter color (1,2,3) '); Read (col);

        Colorm;                 { <---- Set medium res mode     }

        x1 := 16Ø; y1 := 1ØØ;

        x2 := -interval;
        y2 := Ø;
        For count := Ø to xmax do
                [x2 := x2 + interval;   draw(x1,y1,x2,y2,col)];

        y2 := -interval;        x2 := 319;
        For count := Ø to ymax do
                [y2 := y2 + interval;   draw(x1,y1,x2,y2,col)];

        y2 := 199;
        For count := xmax downto Ø do
                [x2 := x2 - interval;   draw(x1,y1,x2,y2,col)];

        x2 := Ø
        For count := ymax downto Ø do
                [y2 := y2 - interval;   draw(x1,y1,x2,y2,col)];
```

```
        Bandw;                  { <---- Return to monochrome    }

        Write ('Do it again? ');        Read (answer);
    Until answer = 'n';
End.
{
-------------------------------------------------------------------
Figure 7-23.    LINDEMO2.PAS    Demonstrates Lines From Center of
                                Screen
*******************************************************************}
```

The BOXDEMO Program

This program is used to demonstrate the use of the BOX routine. Enter the X and Y coordinates for the upper left corner of the box, as well as the dimensions of the box in both the X and Y direction. Notice that this program only draws boxes that are parallel to the X and Y axes. (See Figure 7-24.)

Figure 7-24. Color graphics box demonstration

```
{*****************************************************************
BOXDEMO.PAS     Color Graphics Box Routine.
-----------------------------------------------------------------
}

{$include:'graphics.inf'}       { <---- Interface file          }

Program Boxdemo (input,output);

Uses Graphics;

Var
        x,y,xdim,ydim,col       :integer;
        answer                  :char;

Begin

        Repeat
                bandw;
                Writeln ('Enter x,y,xdim,ydim');
                Readln (x,y,xdim,ydim);

                colorm;
                col := 1;

                Box (x,y,xdim,ydim,col);

                Write ('Again? ');      Read (answer);
```

```
        until answer = 'n';

End.
{
-------------------------------------------------------------------
Figure 7-24.    BOXDEMO.PAS     Color Graphics Box Demonstration
*******************************************************************}
```

Summary

We have scratched the surface of the potential uses for compilands. The color graphics system we've demonstrated is only the start of what could be a wonderful enhancement to your IBM PC Pascal system.

There are parallel systems of programs for just about any application you can imagine. This system approach is especially useful in business applications, where there will be many small tasks to perform in the overall processing scheme.

We have learned how to link code from other sources into a Pascal program, and have covered the use of files to store information, and to transmit it between programs. We have also seen how to break down a complex problem into smaller problems that can be solved individually and then combined for the final solution.

So consider yourself adequately primed to go forward in your accumulation of knowledge and expertise with the IBM PC. You can go on to explore new heights in Pascal programming. There are several other Waite Group books available for the IBM, including an exciting book all about the powerful things you can do with the DOS routines. The tools exist. It's up to you to put them to work.

Exercises

1. What is a library of procedures and functions?

2. When are MODULEs used?

3. How does an INTERFACE file help a programmer?

4. What is the difference between how a procedure in an INTERFACE file is used in a PROGRAM and how it is used in an IMPLEMENTATION?

Solutions

1. A library is a collection of useful procedures and functions that can be used by several different programs.

2. MODULEs are usually used to declare procedures and functions that do not need an INTERFACE file.

3. An INTERFACE provides a single place for complex declarations; the program that uses an interface will have the same declaration as the interface.

4. The procedure in a PROGRAM would be declared as EXTERNAL; in an IMPLEMENTATION, it would be declared as PUBLIC and the body specified.

Appendix—Hexadecimal Numbering

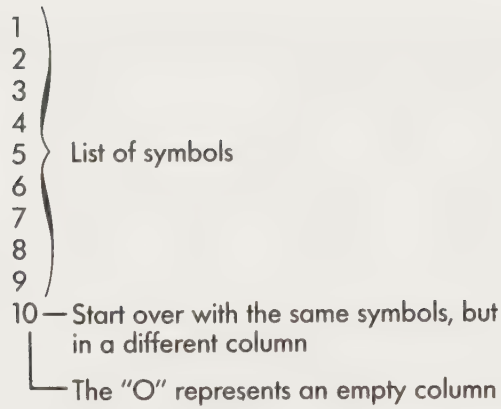
In this appendix we're going to discuss the hexadecimal numbering system. You don't have to be familiar with this system to write simple Pascal programs, but if you want to take full advantage of all of the IBM PC's features, you do need to know something about it. It's also useful in other areas of the computer field — understanding it is a real mark of computer literacy. Unfortunately, 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 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 drawing fifteen little wedges to stand for fifteen sheep was inconvenient.

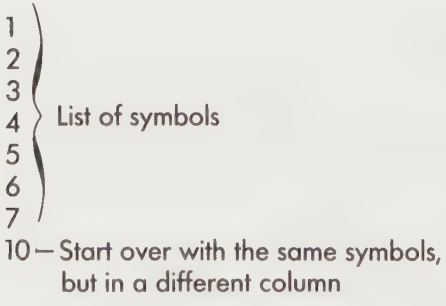
The Roman numbering system was an attempt to streamline things by assigning single symbols to numbers other than one, so that "V" was five, "X" was ten, and so on. However, this system was not completely successful, as generations of school children can attest.

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, and a symbol to stand for nothing, or "zero," is a stroke of genius. It's hard for us to imagine a useful numbering system that doesn't take advantage of these ideas.

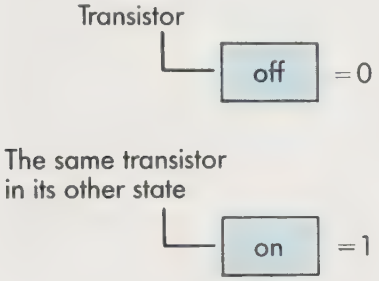
Notice, however, that the number of symbols to be counted before moving to the next column is purely arbitrary. There is no particular reason why there are exactly ten numbers. Well, of course there *is* a reason: the fact that we happen to have ten fingers. But there's no *mathematical* reason. Counting and arithmetic and mathematics would all work just as well with some other number, say eight:



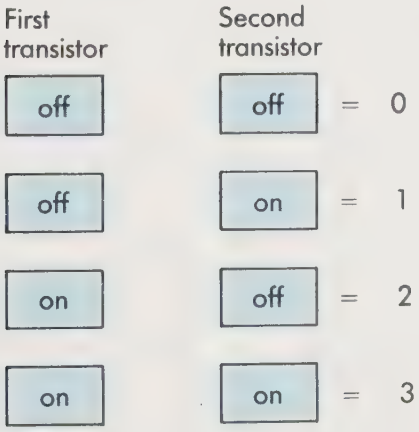
In fact, several numbering systems besides those based on ten have been used in the course of history. The Babylonians favored a numbering system based on sixty. It lingers on, anachronistically, in our clocks and watches; we count up to sixty seconds, then increment the next column to one minute and 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, a one-bit system, with only two possible states.



To represent more than two numbers in a computer we need more transistors. Suppose, for example, we had two transistors. Each of these two transistors can be either 0 or 1, so altogether 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."

00 = 0
01 = 1
10 = 2
11 = 3 — Decimal numbers
└——— Binary numbers

The numbers on the left, which stand for the on or off state of 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:

0 }
1 } List of symbols
10 — Start over with the same symbols, but in a different column

Suppose we had three transistors:

How many things could we count? Let's represent the transistors in binary again:

000 = 0
001 = 1
010 = 2
011 = 3
100 = 4
101 = 5
110 = 6
111 = 7 — Decimal numbers
└——— Binary numbers

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 4 transistors, or binary digits, we can count to 16; with 5 we can count to 32, and so on. Computers frequently use 8 transistors to represent a number, so the number can be as large as 255, as shown below.

```
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
```

Using the binary system makes the transistors in the computer happy: they "think" naturally in terms of binary numbers. And using the decimal system makes us happy: we think naturally in terms of decimal numbers.

If we need to convert from binary to decimal — that is, from computer-counting to human-counting — we can use the table 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 Pascal, 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, and we type in decimal numbers, and it all works out fine. Why can't we do the

same thing in assembly language?

The problem is that when we want to deal more directly with a machine's *hardware* (its memory or input/output devices) 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 numbers. 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.

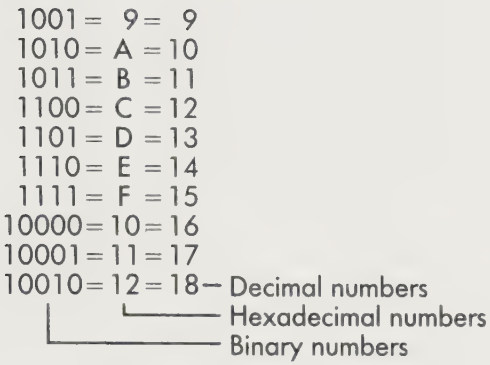
The reason 252 doesn't tell us very much about bit patterns is that each decimal number does not represent a fixed number of transistors (or binary digits). You can't use *three* binary digits to represent the decimal numbers, since with three bits you can only count up to eight. On the other hand if you use *four* binary digits you can count up past ten — to sixteen. There just is no simple relationship between binary and decimal.

What would be ideal 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. (We would use octal if we had a 12-bit or 36-bit computer — multiples of 3 bits — rather than 8-bit and 16-bit computers.) Hexadecimal, or base sixteen, has therefore become the most commonly used computer numbering system.

What exactly does a base sixteen system mean? It means we have 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 another common symbol — 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
```



Notice how four binary digits represent 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 decimal.

Conversions between Binary and Hexadecimal

When you see a number with one hexadecimal digit 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 = 1Ø1Ø1ØØØ,
```

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 = 1Ø11 Ø1ØØ 1ØØ1 1Ø1Ø
```

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

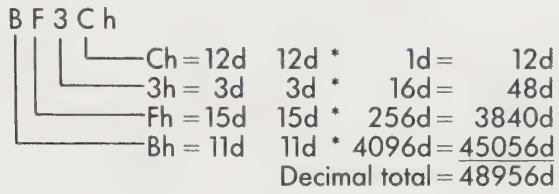
Besides simplifying arithmetic on large hexadecimal numbers, hexadecimal to decimal conversions are often useful in their own right, as you'll find in several places in this book. Assuming you don't have a conversion table, what do you do?

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. (Just as digits in the ten's column in decimal 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. 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).

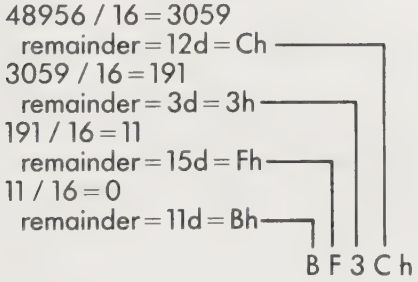


Thus BF3C hexadecimal is 48956 decimal.

Decimal to Hexadecimal Conversion

To do conversions in the other direction, from decimal to hexadecimal, we reverse the process, which involves division by sixteen, 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 hexadecimal digits, from left to right, divide sixteen into the number and use the remainders for each digit. Using our example:



Each step could be performed with the Pascal assignments:

```
digit := num mod 16;
num   := num div 16;
```

You should continue dividing by sixteen until the result (quotient) is zero.

Let's look again at the problem we passed on before:

```
  A84Bh
+ 7C5Fh
  ----
    ?
```

Are you ready to try it? How about if we tell you the answer is 124AA7h. Can you get it?

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 most important 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!

Index

COMPUTER • Z5496 • $17.95
CANADA • $22.50

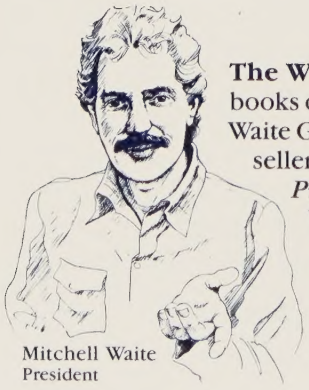
PASCAL PRIMER for the IBM® PC

IBM Enhanced PC Pascal is a powerful language that can be used to create elegant general-purpose programs as well as major systems software. Compatible with PC-DOS, IBM's Pascal offers many advanced features, including compiler directives, attributes, super array type, string processing, constant values, and system implementation.

This book is a primer on the IBM Pascal language. It will take you from the beginning to the advanced stages of PASCAL programming — in short, it's for anyone who wants to develop well-structured software for the IBM PC.

Pascal Primer for the IBM PC discusses the features of IBM Pascal that make it so popular with programmers, and explains, through hands-on examples, the fundamentals of simple variables, program control, structured variables, procedures and functions, pointers and disk files. All the secrets of structured records are revealed. Finally you are shown how to add color graphics and custom assembly routines to your Pascal programs to give your IBM PC unequalled flexibility.

The *Pascal Primer* is completely compatible with other books in the *Plume/Waite* IBM Primer series and enhances the learning process with extensive use of IBM graphics and sound capabilities. Routines from the *Assembly Language Primer* or *Bluebook of Assembly Routines* can be added to your Pascal programs for increased speed and power.



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.

0-452-25496-5