SOFTWARE MANUAL

AlphaPASCAL

USER'S GUIDE

DWM-00100-08

REV. B01

alpha micro

NOTE: This printing of the manual contains the contents of Change Page Packet #1 for the "AlphaPASCAL User's Manual", (DSS-10000-10), which may be ordered separately from ALpha Micro.

First Printing: 1 August 1980
Second Printing: 30 April 1981

This book reflects AlphaPASCAL Versions 2.0 and later.

ALPHA MICROSYSTEMS
17881 Sky Park North
Irvine, CA 92714

Table of Contents

PART IV APPENDICES

APPENDIX A QUICK REFERENCE TO ALPHA PASCAL

CHAPTER 1

INTRODUCTION

This book is a reference manual for the AlphaPascal programming system. We realize that some of you may be experienced Pascal programmers, while others may never have seen a Pascal program before. Therefore, to suit the wide range of interests and backgrounds our readers are likely to have, we have tried to organize this book so that you can easily find the information that you need without spending unnecessary time on chapters that contain information that you already know or that is not important to you. (For information on the organization of this book, see Section 1.1, below.)

Because there are so many excellent books available that teach you how to program in Pascal, we have not attempted to do so in this book. (For a list of some of the books that we found helpful, see Section 1.2, "Pascal Bibliography.") However, our intention is to provide a detailed enough description of AlphaPascal that an experienced computer programmer who is unfamiliar with Pascal can get some idea of how to write Pascal programs.

The major purpose of the book is threefold:

1. To describe this implementation of AlphaPascal;

2. To discuss how this implementation differs from previous versions of AlphaPascal and from the standard Pascal as set forth in the Pascal User Manual and Report by Jensen and Wirth (and to give hints on converting programs written in these versions of Pascal to the current AlphaPascal format); and

3. To give operating instructions for the various components of the AlphaPascal programming system: the compiler, the linker, and the run-time package.

This book also gives information to systems programmers on writing their own assembly language subroutines callable by Pascal programs, and on writing and modifying an external procedure library.

1.1 ORGANIZATION OF THIS BOOK

Some of the chapters in this book are aimed at experienced Pascal programmers, while others are specifically for new Pascal users. To help you find the information that you are particularly interested in, we have divided this book into four general parts:

PART I - THE ALPHA PASCAL SYSTEM

PART II - SUMMARY OF ALPHA PASCAL

PART III - ADVANCED PROGRAMMING ON THE ALPHA PASCAL SYSTEM

PART IV - APPENDICES

The rest of this section discusses which chapters may be of particular interest to specific readers.

IF YOU ARE AN EXPERIENCED PASCAL PROGRAMMER:

You will probably want to skip Chapter 2, "Getting Started," and go directly to Chapter 3, "Compatibility and Conversion," which tells you how this version of Pascal differs from earlier versions of AlphaPascal and from the Jensen and Wirth standard. Chapter 4 discusses how to operate the various components of the AlphaPascal programming system. Rather than read through Chapters 5 through 13, which give detailed discussions of the AlphaPascal statements and procedures, you may want merely to turn to Appendix A, "A Quick Reference to AlphaPascal," to get an idea of the functions and procedures included in this implementation of Pascal.

After you are somewhat familiar with the AlphaPascal system, you may want to read Chapter 15, "Writing and Modifying an External Library File." If you are a systems programmer, you may want to read Chapter 16, "Assembly Language Subroutines."

IF YOU ARE NEW TO PASCAL:

You will probably want to read Chapter 2, "Getting Started," which gives a brief discussion of Pascal, and goes through a quick demonstration of building, compiling, and running a small, simple Pascal program. Next, you will probably want to start reading Part II, "Summary of AlphaPascal," for information about this version of the Pascal language.

When you are ready to begin writing Pascal programs, turn back to Chapter 4, "Operating Instructions and Characteristics," for information on using the AlphaPascal compiler and run-time package.

NOTE: We would appreciate any comments or suggestions; note the Reader's Comments Form in the back of this book.

1.2 PASCAL BIBLIOGRAPHY

The most important source book for Pascal programmers (containing the definition of standard Pascal) is:

Jensen, K. and Wirth, N.
Pascal User Manual and Report (Second Edition)
Springer-Verlag, 1976

If you are interested in learning to program in Pascal, you might want to take a look at one or more of the following textbooks:

Conway, R., Gries, D. and Zimmerman, E.C.
A Primer on Pascal
Winthrop, 1976

Grogono, P.
Programming in Pascal
Addison-Wesley, 1978

Kieburtz, R.B.
Structured Programming and Problem-Solving with Pascal
Prentice-Hall, Inc., 1978

Schneider, G.M., Weingart, S.W., and Perlman, D.M.
An Introduction to Programming and Problem Solving with Pascal
John Wiley & Sons, 1978

Wilson, I.R. and Addyman, A.M.
A Practical Approach to Pascal
Springer-Verlag, 1978

1.3 GRAPHICS CONVENTIONS USED IN THIS BOOK

The symbol (RET) indicates the place in an example where you would press the terminal carriage return key if you were entering the example into the computer. (The carriage return key on the terminal keyboard is usually labeled RET or RETURN, and tells the computer to accept and process the current line.)

It is often confusing when looking at a program in a new computer language to determine which elements are an inherent part of the language (for example, program statements) and which elements are to be supplied by the programmer. To help eliminate some of this confusion, our sample programs follow these conventions:

Reserved words are all upper case and underlined.

Standard identifiers are all upper case, but not underlined.

All user identifiers (for example, variable names, constants, etc.) are in a combination of upper and lower case, and are not underlined.

(Note that reserved words are underlined. For clarity's sake, therefore, this manual deviates from the usual Alpha Micro documentation practice of underlining all output the computer displays on your terminal display. We will try to clearly indicate which portions of our examples are entered by you and which portions are printed by the computer.)

CHAPTER 2

GETTING STARTED

This chapter is primarily for the benefit of the programmer who is interested in learning Pascal, but who has not yet had the chance to become familiar with the language. If you are already familiar with Pascal, you will probably want to skip to Part I of this book, "The AlphaPascal System," for information on Alpha Micro's specific implementation of Pascal, and for operating instructions for the AlphaPascal compiler. (You may be interested in Section 2.3 of this chapter, however, which contains a brief demonstration of creating, compiling, linking, and running a small Pascal program.)

The rest of this chapter gives a brief discussion of Pascal and walks you through a quick demonstration of building, compiling, linking, and running a program under the AlphaPascal system.

We also show you a small Pascal program and discuss its component parts.

2.1 WHAT IS PASCAL?

The Pascal language is based on the 1970 work of Jensen and Wirth, and is related to the ALGOL-family of languages.

Pascal is a fairly new programming language, and is considered by many to be "cleaner" and more powerful in design than many older languages as well as more reflective of current trends in the philosophy of program design and structure. However, this does not mean that programs written in Pascal will necessarily be clearer or more powerful than programs written in other languages-- that will depend on the programmer. The major claim made for Pascal is that the language makes it easier to write programs that may be easily understood and maintained.

It was developed in response to increasing concerns that current programming languages were not encouraging good programming "style," and is based on the idea that an effective programming language should help the programmer to apply design techniques in a natural and simple way. The result should be well-made, well-structured programs that are easy to read and easy to

maintain. Because most of a program's life cycle is spent in design and maintenance, the creators of Pascal tried to develop a language that helps programmers in these areas.

Pascal's use, acceptance, and availability have become widespread in recent years. An increasingly large number of students are being taught Pascal as their first programming language. Pascal's use in industry is also becoming more prevalent as project planners become more aware of its usefulness in implementing large programming projects.

Some of Pascal's advantages stem from these characteristics:

* Pascal encourages well-structured programming by requiring that programs be built in a block structure in which the beginning and end of each procedure is clearly marked. Because program structure is hierarchial in nature, programming in Pascal lends itself naturally to top-down design.

* One of the most important features of Pascal is its extensibility. It is very simple to add your own functions and procedures if the routines provided by Pascal do not exactly match your needs. In addition, on the AlphaPascal system, you can add these user-defined routines to an external library where all Pascal programmers can make use of them.

* Pascal was designed to be a general-purpose language. Since it is not specifically aimed at scientific or data processing applications, it can be used to solve a wide range of problems.

* An important feature of Pascal is its powerful data structures (arrays, sets, records, pointers, user-defined, etc.), and the sophisticated structures you can build from those primitives (e.g., linked lists).

* Any variable used in a program must be declared within that program. That is, Pascal requires that the type of values that a variable may assume (e.g., integer or boolean) be clearly stated by the programmer. This helps both in program design and maintenance, since the readability and organization of your program are enhanced. Variables may be global or local in scope, depending on where they are declared.

* Most implementations of Pascal, while they may include extensions to the language, also contain a subset of Pascal which adheres firmly to the standards for the language as set forth by Jensen and Wirth. This means that programs written in standard Pascal are transportable between computer systems on which Pascal is implemented.

2.2 SAMPLE PROGRAM

If you have never before seen a Pascal program, you may be interested in taking a look at the small, simple program below:

```
{Determine what % is deducted from your gross salary}
PROGRAM Salary;
VAR   Gross, Takehome, Deductions, Percentage : REAL;
BEGIN {Begin Program Salary}
   {Print questions and read answers from terminal}
   WRITE ('What is your gross salary? ');
   READLN (Gross);
   WRITE ('What is your takehome salary? ');
   READLN (Takehome);
   Deductions := Gross - Takehome;
   Percentage := 100*(Deductions/Gross);
   WRITELN ('They keep', Percentage, ' percent of your salary!')
END {End Program Salary}.
```

NOTE: To help you keep track of which words in the program are elements of the Pascal language and which are variable names and data supplied by you, we have written in upper case and underlined those words (called "keywords") that are actually part of the Pascal language. (Of course, you do not underline such keywords when you write your own Pascal programs.) Those words that are in upper case, but that are not underlined, are called "standard identifiers"; they are elements of the Pascal language which can be re-defined by you. The words that are upper and lower case and that are not underlined in the example above are variable names, comments, and string data supplied by the writer of the program.

The first line of our sample program is called a "comment." It is ignored by the computer, and has no effect on the execution of the program. Its purpose is to make the program easier to read for humans. (Comments in Pascal are denoted by enclosing text either with the symbols "(*" and "*)" or with the symbols "{" and "}".)

The second line "declares" the program name, "Salary."

The third line "declares" the variables "Gross," "Takehome," "Deductions," and "Percentage," and tells Pascal that they can only assume the values of real numbers. (For information on declaring programs and variables, see Chapter 6, "Declarations and Definitions.")

The fourth line contains a BEGIN statement; this statement marks the beginning of a program block. The end of this block (and in this case, the end of the program) is marked by the END statement on line 13. Within this block, we send questions to the terminal display (sixth and eighth lines) and read data from the terminal keyboard (seventh and ninth lines). On the tenth and eleventh lines we compute the answer we need based on the data we received from the user of the program. The twelfth line sends the computed answer to the terminal display. (For information on Pascal program statements and procedures, see Part II, "Summary of AlphaPascal.")

(NOTE: So that we could identify specific lines of the program to you, we mentioned identifiers such as "first line" or "fourth line." This was for our convenience only; the lines in Pascal programs do not ordinarily start with numbers.)

2.3 BRIEF DEMONSTRATION

Now that you've taken a look at a small Pascal program, we would like to walk you through a brief demonstration of building, compiling, linking, and running the program.

We'll assume that the computer and your terminal are on, and that you have been assigned an account in which to work. Make sure that you are at AMOS command level (that is, that you see the prompt symbol, ".", that indicates that you are communicating with the operating system).

First, log into the system by typing LOG followed by the device that contains the account you want to log into and then entering the number of that account. Then press the RETURN key on your terminal. For example, if you want to work in account [20,3] on device DSK1:, enter:

```
LOG DSK1:[20,3] (RET)
```

Now you see something like:

```
Logged into DSK1:[20,3]
```

You can now begin to create your Pascal program.

2.3.1 Building a Pascal Program

To build a Pascal program, use one of the system text editors to create your program as a text file. If you are using a video-display terminal, you will probably want to use the screen-oriented text editor, VUE, rather than the character-oriented text editor, EDIT.

2.3.1.1 The VUE Text Editor - First, we'll make sure that no earlier versions exist of the program we're going to create. So, we'll erase from the disk any file called SALARY.PAS. At AMOS command level, enter:

```
ERASE SALARY.PAS (RET)
```

If you see:

```
SALARY.PAS erased
Total of 1 file deleted, 2 disk blocks freed
```

that means that the file did exist on the disk, and that we have now erased it. If you see:

```
%No files erased
```

no error occurred, it's just that no file named SALARY.PAS was in the account you are logged into, and so we couldn't erase it. In either case, you now are free to create a new file of the name SALARY.PAS.

So, enter:

```
VUE SALARY.PAS (RET)
```

Now VUE looks for the disk file SALARY.PAS in the account you are logged into. Since the file does not yet exist, VUE says:

```
SALARY.PAS does not exist - create it?
```

Enter a Y followed by a RETURN to tell VUE that you do want to create a new file named SALARY.PAS.

Now you see one or more lines of asterisks. (If you do not see this display, but instead see a display whose first line begins: "AlphaVue n.n Status:" (where n.n is the version number of VUE), simply type an Escape (sometimes labeled ESC or ALT MODE on your keyboard), and VUE will display the asterisks.)

The display of asterisks means that you are in editing mode, and that VUE is ready for you to type your program in. Start typing the sample program in Section 2.2 just as you would if you were using a typewriter. Type in the example exactly as shown, including all semicolons, quote marks, and parentheses.

If you make a mistake, you may erase single characters by using the RUB key (sometimes labeled DEL or DELETE). To erase the characters on an entire line, type a Control-RUB. (That is, hold down the CONTROL key while you press the RUB key.)

The cursor (which may appear as a small white rectangle, triangle, line, or other symbol) marks your place on the screen; the next character you type appears at the cursor position. If more extensive corrections are needed, you may back up in the display by using the arrow-keys to move the cursor back and forth in the text on the screen. (If your terminal does not have these arrow-keys, you must move the cursor by typing Control-J, Control-H, Control-K, and Control-L. For example, to move the cursor to the left, hold down the CONTROL key and type an H.)

When the cursor is positioned just to the left of the error, you can overwrite the error by typing your new characters over the problem spot. Or, if you do not want to overwrite the error, type a Control-Q. From this point on, the new characters you type will be inserted into the current line, rather than overwriting it. (To resume overwriting characters, type another Control-Q.)

Of course, there are many more VUE editing commands that we won't discuss here. You can, for example, erase characters a word at a time, insert entire new lines of text, search for particular groups of characters, or move the cursor a word at a time. For more information on using VUE, see the AlphaVUE User's Manual, (DWM-00100-15).

When the program is entered correctly, you are ready to leave VUE. Type an Escape. The screen clears, and the cursor is now positioned next to the VUE prompt symbol, >. (You are now in command mode.) Type an F followed by a RETURN. This tells VUE that you are finished; it therefore writes your file SALARY.PAS out to the disk. Next you see the AMOS prompt symbol, a dot, which tells you that you have exited VUE, and are now back at AMOS command level.

Here is a summary of the keys that you will use the most when editing programs with VUE:

RETURN — End each line with a carriage return symbol by pressing the RETURN key (sometimes labeled RET, CR, or CARRIAGE RETURN).

ESC — To change from editing mode to command mode (and back again), type an Escape by pressing the ESC key (sometimes labeled ALT MODE or ESCAPE).

CONTROL — Most of the VUE commands are control-characters. To type a control-character, hold down the CONTROL key (sometimes labeled CTRL), and type the appropriate character. For example, to type a Control-C, hold down the CONTROL key while you type a C.

RUB — To delete a single character, press the RUB key (sometimes labeled DELETE or DEL).

ARROW-KEY — To move the cursor around on the screen, use the keys marked with arrows (labeled with a left-arrow, right-arrow, up-arrow, and down-arrow). For example, to move up on the screen, press the up-arrow key. If your terminal does not have arrow keys, you will use these control-characters instead:

Control-H	To move left
Control-J	To move down
Control-K	To move up
Control-L	To move right

If VUE is new to you, you may want to ask the System Operator to place into your account a copy of the VUE initialization file in which the menu-display option has been enabled. VUE will then display a summary of its commands when you enter command mode. You may also want to ask the System operator to modify the VUE initialization file so that the default extension is set to .PAS (which means that VUE will expect you to edit .PAS files and thus will not require you to enter a file's extension unless you want to edit a non-.PAS file).

2.3.2 Compiling and Linking a Pascal Program

The first step after creating your program is to compile it using CMPILR. After you have compiled it, the program is still not ready to run until you use the linker, PLINK. (Both PLINK and CMPILR are themselves programs written in Pascal.) Chapter 4, "Operating Instructions and Characteristics," discusses the operation of PLINK and CMPILR in detail. For now, we'll simply show you one way to use them-- to compile and link a new program made up of only one file. For this demonstration, we will use one of the command files provided with your system, PCL.DO. This command file contains a series of commands and data that automatically invoke CMPILR and PLINK for you, and provide necessary information to those programs. NOTE: Remember that the larger your memory partition is, the faster your programs will compile!

At AMOS command level, enter PCL followed by the name of your program (leaving off the .PAS extension). Then type a RETURN. For example:

```
PCL SALARY(RET)
```

Now the PCL command file runs CMPILR and PLINK for you. As your program is compiled, you see a display something like this:

```
PRUN CMPILR
AlphaPascal V2.0
Source file name? SALARY
Diagnostic file name (<return> for terminal)?
AlphaPascal Compiler Version 2.0
          <   0>---
 PROGRAM  <   3>--------
 12 lines
 10.47 seconds, 68.79 lines/minute
 No compilation errors.
```

If CMPILR spots an error while it is compiling your program (for example, if we left the semicolon off the end of the second line), CMPILR pauses, and tells you about the problem. For example:

```
VAR    Gross, Takehome, Deductions, Percentage : REAL;
^

?Line 2: [INISOP] ';' or ')' expected -- inserting ';'
Hit RETURN to continue
```

The message above tells us that a semicolon is missing in front of the symbol VAR. ("[INISOP]" identifies the portion of the compiler that caught the error-- you can disregard that information.)

Now you may type a RETURN to resume program compilation, or you may type a Control-C (hold down the CONTROL key while you type a C) to interrupt the compilation. (If you type a Control-C, CMPILR displays the message: "?Compilation aborted" and then returns you to AMOS command level. If you type a RETURN, CMPILR resumes the compilation, and then returns you to AMOS command level. In either case, because an error has occurred PCL does not go on to link the program and you are returned to AMOS command level.)

If CMPILR reported something other than "No compilation errors," your program is incorrect. You should use VUE on the program and check your copy

of the program against the one in this book. Correct any discrepancies, and use the PCL command file again. (For full information on using CMPILR and its options, refer to Section 4.3, "The AlphaPascal Compiler." That section also discusses the compiler display.)

Let's say that your program has compiled without error. PCL.DO goes on to invoke the linker, PLINK. At this point, CMPILR has created three intermediate files: SALARY.P01, SALARY.P02, and SALARY.P03. However, your program still is not completely ready to run. PLINK will fully resolve references within the intermediate files and will produce the final, executable .PCF file. The second part of the screen display that you see looks something like this:

```
.ERASE SALARY.PCF
%No files deleted
.PRUN PLINK
AlphaPascal V2.0
Code file = SALARY
Creating new code file SALARY.PCF
Library code file for SALARY.PCF = STDLIB

Please specify files to be linked into SALARY,
 one per line, ending in a blank line

   File 1 = SALARY
   File 2 =
Loading program and library dictionaries
Processing SALARY
  Linking in global func/proc PROGRAM
Transferring temporary file to new code file
SALARY completed
```

The first thing that the command file does before linking your file is to erase any file SALARY.PCF that already exists. (This is because PLINK asks different questions depending on whether or not the specified program already exists, and we want to make sure that PLINK asks a particular set of questions.) Now it invokes PLINK.

For more information on linking a file, see Section 4.4, "The ALphaPascal Linker." That section also discusses the meaning of the display you see above, and talks about the concept of a "library."

2.3.3 Running a Pascal Program

To run the program you have compiled, use the Pascal run-time package, PRUN. At AMOS command level, enter:

```
PRUN SALARY.PCF (RET)
```

followed by a RETURN. At last your program is running! (For full information on executing Pascal programs, refer to Section 4.5, "The

AlphaPascal Run-time Package.")

As you run SALARY.PAS, you see:

```
AlphaPascal V2.0
What is your gross salary?
```

Let's assume that you want to enter 250 as your gross salary and 175 as your takehome. Below is a sample run of your program:

```
AlphaPascal V2.0
What is your gross salary? 250 (RET)
What is your takehome salary? 175 (RET)
They keep 30 percent of your salary!
```

PART I

THE ALPHA PASCAL SYSTEM

The next two chapters introduce you to the AlphaPascal programming system. Chapter 3 is aimed at the experienced Pascal programmer; it discusses the differences between this implementation of Pascal and previous versions of AlphaPascal. It also discusses the major differences between this Pascal and the standard Pascal as described in Jensen and Wirth's Pascal User Manual and Report. The last section of Chapter 3 gives some hints for converting programs written in earlier versions of AlphaPascal over to the current AlphaPascal standards.

Chapter 4 gives full operating instructions for the various components of the AlphaPascal system; the compiler, the linker, and the run-time package. Chapter 4 tells you everything you need to know about the actual processes of creating, compiling, linking, and running an AlphaPascal program. Chapter 4 also discusses file requirements and memory limitations of the AlphaPascal system.

CHAPTER 3

COMPATIBILITY AND CONVERSION

This chapter is aimed primarily at the experienced Pascal programmer who wants to know how this implementation of Pascal differs from previous versions of AlphaPascal and from the standard Pascal described by Jensen and Wirth in the Pascal User Manual and Report.

We have also included a section that provides hints on converting Pascal programs written under earlier versions of AlphaPascal to the format used by the current AlphaPascal.

If you have never before programmed in Pascal, you will probably want to skip this chapter and go directly to Part II, "Summary of AlphaPascal," for information on the Alpha Micro Pascal, or to the next chapter, "Operating Instructions and Characteristics," for information on using the AlphaPascal compiler and linker.

3.1 PREVIOUS VERSIONS OF ALPHA PASCAL

Previous versions of AlphaPascal were based on the UCSD Pascal programming system, Version I.4. In order to provide a Pascal that is more fully integrated with the Alpha Micro operating system and file system, we now offer this new version of AlphaPascal that was expressly developed for the Alpha Micro computer.

To make life easier for programmers who have written programs using previous versions of AlphaPascal, we have tried to keep many of the same features and functions, while adding a number of new extensions and abilities. Most of the changes between this version and earlier versions are added features that do not require that you rewrite your earlier programs.

Several of the most important difference are:

- The operating instructions for AlphaPascal have changed. An important difference is that you will use the Alpha Micro screen-oriented text editor, VUE, to create your programs. You must also use the linker,

PLINK, to link any compiled program, whether or not it consists of more than one file. See Chapter 4, "Operating Instructions and Characteristics," for complete instructions.

- Expression handling has been considerably enhanced:

1. You may now include the assignment operator in an expression. For example:

```
5 + X := 7
```

The expression above is equivalent to 5 + (X := 7), and means "Let X assume the value of 7, and then be added to 5."

2. Wherever an expression is legal, you may include an IF-THEN expression of the form:

```
IF condition THEN expression ELSE expression
```

For example:

```
Year := (IF Feb = Leap THEN 29 ELSE 28)+337;
```

If Feb equals the value Leap, then Year assumes the value 29+337; otherwise it assumes the value 28+337.

3. Wherever an expression is legal, you may include a CASE expression of the form:

```
CASE value OF value1 : expression;
              value2 : expression;
                    .
                    .
              ELSE expression
```

For example:

```
WRITE(CASE Errorcode OF
        1 : 'Illegal input';
        2 : 'Number too large';
        3 : 'Number too small';
        ELSE 'undefined error');
```

- AlphaPascal now recognizes modifying assignment operators. These operators are:

```
+=      Adding assignment operator
-=      Subtracting assignment operator
*=      Multiplying assignment operator
/=      Dividing assignment operator
```

These operators tell the compiler to modify (instead of replace) the value of the variable on the left of the assignment operator with the value of the expression on the right of the operator.

For example, in the case of the adding assignment operator:

```
X += 1
```

tells the compiler to let X assume the value of X+1. For more information on these operators, see Section 8.1.2.1, "Modifying Assignment Operators."

- Operator precedence has been changed to make it more compatible with operator precedence in other language processors on the Alpha Micro system. The relational operators have been made of higher precedence than the Boolean operators. (See Section 8.1.1, "Operator Precedence," for more information.)

- AlphaPascal allows you to label BEGIN-END blocks by following the BEGIN and END keywords with a colon followed by an indentifier. These labels allow you to tell the compiler which BEGINs and ENDs should match. If the structure of your program is such that they do not match, the compiler will tell you so.

For example:

```
BEGIN : Block1
          .
          .
  BEGIN : Block2
          .
          .
  END : Block2
          .
          .
END : Block1
```

The compiler checks these labels to make sure that the designated pairs of BEGIN-END keywords are indeed properly matched. For example, the following program would cause an error because the BEGIN-END blocks are not properly nested:

```
BEGIN : Block1
          .
          .
  BEGIN : Block2
          .
          .
END : Block1
          .
          .
  END : Block2
```

- Two new keywords have been added to AlphaPascal: EXTERNAL and MODULE. These words may no longer be used as identifiers. If they do appear in your programs, you see an error message (e.g., "[TRYSCAN] VAR, PROCEDURE, or FUNCTION expected -- scanning") when you compile the programs.

  EXTERNAL allows you to access variables, procedures, and functions in an external library, and allows a file in a multiple-file program to access variables, procedures, and functions in another file. See Section 6.7, "External Declarations," for more information.

  The MODULE keyword designates a file that does not contain the main program portion of the program. Modules may contain declaration and definition statements, but may not contain the final BEGIN-END block. (That is, BEGIN-END blocks may only appear in function or procedure definitions if they appear in modules.) See Section 6.1, "Program Declarations," for more information.

- The SEGMENT keyword and segment procedures are no longer supported. (See the discussions of EXTERNAL and MODULE, above.) Remove the SEGMENT keyword from your programs.

- Floating point numbers are now three words in length (i.e., 12 digits). (They used to be two words, and could only represent six digits.)

- You may call assembly language subroutines from within your Pascal programs. For information on writing assembly language subroutines, see Chapter 15, "Assembly Language Subroutines."

- Opening, closing, and specifying files have changed. You may now access AMOS files, and make full use of the Alpha Micro file system. Refer to Chapter 10, "Input/Output Functions and Procedures," for more information on the procedures and functions that allow you to search for, open, and read and write sequential and random files. (NOTE: Those of you who have done assembly language programming using monitor calls on the AMOS system will recognize some of the new procedure names such as FSPEC, OPEN, OPENI, OPENO, and OPENR.)

- AlphaPascal supports an external procedure library. This library contains a series of procedures and functions available to your programs. You may write your own external libraries that make use of the library provided. See Section 16.1, "STDLIB," for a list of procedures and functions in the library. If you wish to access these routines in your programs, your programs may not use these names in global identifier definitions, since such definitions will override the standard library definitions.

  If you wish to access these procedures and functions, simply invoke them in your program. If they are not defined within that program, AlphaPascal assumes that they are in the external library.

- Several procedures and identifiers used by previous versions of AlphaPascal are not supported by the current version:

  BLOCKREAD
  BLOCKWRITE
  UNITREAD
  UNITWRITE
  UNITWAIT
  UNITBUSY
  UNITCLEAR
  GOTOXY (Refer to Section 11.2.2, "CRT," for information on cursor positioning.)
  HALT
  IORESULT
  INTERACTIVE files

- PROGRAM (the main program) may not be called recursively.

- You should be aware of these changes to the standard procedures:

  1. RESET and REWRITE accept only one argument: a variable of type FILE. You may not specify a filename after that argument.

  2. The file type INTERACTIVE is no longer supported or needed; replace it with the standard file type TEXT.

  3. In earlier versions of AlphaPascal, CLOSE took an option as an argument in addition to a variable of type FILE; it now accepts only a single argument-- a variable of type FILE.

  4. When you use the EXIT statement to exit a program, you must supply the PROGRAM keyword as the argument, not the program-name. (That is, EXIT(PROGRAM) is valid, but EXIT(NewProgram) is not.) You may, however, exit a procedure or function by giving the name of that procedure or function (e.g., EXIT(EvalError)).

  5. WRITE and WRITELN do not accept a Boolean variable as an argument. That is, if NewFile is a Boolean variable which evaluates to TRUE:

     ```
     WRITELN(NewFile);
     ```

     does not print TRUE, but instead generates an error.

3.2 STANDARD PASCAL

The standard Pascal is described by Jensen and Wirth, in the Pascal User's Manual and Report (Second Edition). AlphaPascal differs from this standard in several ways (also, note the extensions discussed in Section 3.1, above):

- The program heading file identifiers are scanned but ignored. That is, if you have any information in the program heading after the program name, that information is ignored. (For example, "PROGRAM MailBox;" is equivalent to "PROGRAM MailBox(INPUT,OUTPUT);".) This is because AlphaPascal uses its own form of file handling that is consistent with the AMOS file structure. (Note, however, that the remainder of the heading after the program name is scanned, and that therefore the program heading must be syntactically correct. For example: "PROGRAM NewAccount (;" will generate an error because of the open parenthesis.) If you want to use any files other than the predeclared file INPUT and OUTPUT, you must use VAR statements to declare them.

- Operator precedence has been changed to make it more compatible with other language processors on the Alpha Micro system. If it is important that your program be able to run under another Pascal that uses standard Pascal's rules of operator precedence, you will have to use parentheses in your expressions to override AlphaPascal's rules of operator precedence.

  This will only become necessary if your expressions use relational operators to compare Boolean expressions. For example, if A, B, C, and D are Boolean variables, standard Pascal evaluates: IF A = B AND C = D THEN... as: IF (A = (B AND C)) = D THEN ..., while AlphaPascal evaluates it in this way: IF (A = B) AND (C = D) THEN...

  (See Section 8.1.1, "Operators," for information on operator precedence.)

- Two new keywords have been added to the list of reserved words: EXTERNAL and MODULE. In addition, several identifiers have been added to the standard identifier list. (For a list of AlphaPascal standard identifiers, see Section 5.4.2, "Standard Identifiers.")

  Also, several standard identifiers used by standard Pascal are NOT used by AlphaPascal (DISPOSE, PACK, and UNPACK) since AlphaPascal does not use these procedures. AlphaPascal uses MARK and RELEASE to reclaim memory allocated by NEW, and automatically unpacks packed data structures for you when necessary. (See Section 11.1.4, "NEW," for information on allocating dynamic variables.)

- Standard Pascal supports the data type CHAR (single character). AlphaPascal also supports a non-standard type, STRING, which contains a length field as well as a field of characters. (See Section 7.2.3, "STRING," for a description of this data type.)

3.3 MAKING PROGRAMS COMPATIBLE WITH THE NEW ALPHA PASCAL

In general, programs written in previous versions of AlphaPascal or standard Pascal will require very little modification before being runnable under the current AlphaPascal. For example, the sample program given in Chapter 2 runs correctly in any of these versions of Pascal. The largest number of changes will probably involve functions and procedures that read and write disk files, since the new AlphaPascal is fully integrated into the AMOS file structure.

If your programs were written under previous versions of UCSD/AlphaPascal, you will need to transfer your programs to AMOS files before you begin to perform any necessary conversions. To do so, use the UCSD/AlphaPascal programming system (which was provided only in earlier releases of AlphaPascal):

1. At AMOS command level, enter the UCSD/AlphaPascal programming system by typing PASCAL followed by a RETURN:

   PASCAL (RET)

   When you see the initial prompt:

   Command:E(dit,R(un,F(ile,C(ompile,X)ecute,D(ebug,I(nit,H)alt

   Type an F.

2. You are now communicating with the Filer. You see this prompt:

   Filer:G(et,S(ave,W(hat,N(ew,L(dir,R(em,C(hng,T(rans,D(ate,Q(uit

   To see what is in your library, type L. Now you see the question:

   What volume?

   Enter a colon followed by a RETURN. Now you see a list that might look something like this:

   ```
   SCR:
   ROMAN.TEXT      4       28-Jun-80
   POSTFIX.TEXT    4       28-Jun-80
   2 files, 8 blocks used, 26 unused
   ```

   This is a list of the files in your library.

3. You see the Filer prompt again. To write one of the programs out to an AMOS file, enter T.

   a. The Transfer function asks you:

      Transfer what file?

      enter one of the files listed in the directory. For example:

Transfer what file? ROMAN.TEXT (RET)

b. Now Transfer asks:

To what file?

Enter "REMOTE:" and type a RETURN:

To what file? REMOTE: (RET)

c. Now Transfer asks:

Using what AMOS file?

Enter a valid AMOS file specification. For example:

DSK1:CONVRT.PAS

IMPORTANT NOTE: You must make sure that this file does not already exist; if it does, the UCSD/AlphaPascal system will not do the transfer, and will make the accessed drive inaccessible to you (that is, it will declare that drive "off line") until you exit or re-enter Pascal.

d. Now Transfer asks:

CONVRT.PAS mode: T(ext, I(mage:

Enter an upper case T followed by a RETURN.

e. Now transfer begins to copy ROMAN.TEXT into the AMOS file DSK1:CONVRT.PAS. When Transfer is done, you see:

SCR:ROMAN.TEXT transferred to REMOTE:

You may now use the text editor, VUE, to modify the AMOS file that contains your program. NOTE: If your file is too large, Transfer may ask for additional AMOS file specifications. When you are finally finished, you will need to append all such files into a single file, using the AMOS APPEND command.

Here is list of things to check when converting your old programs to current AlphaPascal format:

1. Make sure that you do not use the reserved words EXTERNAL or MODULE as identifiers.

2. Check the list of standard identifiers in Section 5.4.2, "Standard identifiers," to make sure that you do not redefine any identifiers that designate functions or procedures you need by including them in global declarations.

3. Remove any information concerning input or output files from your program heading.

4. The INTERACTIVE file type is no longer supported. Change any occurrences of the INTERACTIVE file type in your programs to TEXT. It might be easiest to just redefine INTERACTIVE at the front of your programs via a type statement:

```
TYPE  INTERACTIVE = TEXT;
```

5. Previous versions of AlphaPascal expected a UCSD file specification for the argument of the compiler include option, $I. Now the $I option request accepts an AMOS file specification. The default extension is .INC. If you have used the $I compiler option, you will have to change your file specifications to valid AMOS file specifications, and make sure that those files exist. For more information on include files, see Section 4.3.2.2, "The Include Option ($I)."

6. If it occurs in your programs, remove the SEGMENT keyword.

7. Note that the operator precedence used by AlphaPascal is different from that of standard Pascal and previous versions of AlphaPascal. You may need to check expressions in which Boolean expressions are compared with relational operators to make sure that the expressions will be evaluated correctly. See Section 8.1.1, "Operator Precedence," for more information.

Besides changing your programs so that they will run under AlphaPascal, you might also want to add some of the new AlphaPascal features listed in Section 3.1, above. As an example, instead of the statement:

```
TOTAL := TOTAL + SUM;
```

you might want to say:

```
TOTAL += SUM
```

Or, you may want to break your programs up into modules. (For information on modules, see Section 6.1, "Program Declarations.") Of course, if you want your programs written in standard Pascal so that they can run with other Pascal implementations, you may want to restrict your programs to using features found only in standard Pascal.

CHAPTER 4

OPERATING INSTRUCTIONS AND CHARACTERISTICS

This chapter assumes that you are ready to start compiling and running Pascal programs. If you are not familiar with AlphaPascal, you may want to skim through Part II, "Summary of AlphaPascal," before you attempt to start using the AlphaPascal system. This chapter gives you information that you will need to know about the programs that make up the AlphaPascal programming system. The first few sections talk about file and memory requirements. Operating instructions begin with Section 4.2, "Creating a Pascal Program."

The AlphaPascal system consists of the compiler, CMPILR; the linker, PLINK; the run-time package, PRUN; and, the standard external library, STDLIB.

To create a Pascal source program, use the system screen-oriented text editor, VUE. VUE is an easy to use, powerful editor that allows you to see your Pascal program on the screen of your terminal, and to make changes to that program by moving the cursor around on the screen display and entering the new or replacement characters. For information on using VUE, see the AlphaVUE User's Manual, (DWM-00100-15). (Also, a brief introduction to VUE is given in Section 2.3.1 of this book, "Building a Pascal Program.")

After creating your program, you will exit VUE and use the AlphaPascal compiler, CMPILR, which compiles your source program (a file that has the .PAS extension) into a series of intermediate files. Next you will use the AlphaPascal linker, PLINK, which uses the intermediate files created by the compiler to create a fully resolved, runnable P-code file that has the .PCF extension. The linker also allows you to link together separate files into one program, and allows you to update one portion of an existing compiled program without re-compiling all of the modules that make up that program. To run your .PCF file, you will use the AlphaPascal run-time package, PRUN.

The external library contains a set of procedures, variables, and functions that are available to your Pascal programs. For a list of the routines within the external library, see Section 16.1, "STDLIB." For information on writing and modifying your own procedures within this library, see Chapter 16, "Writing and Modifying an External Library File."

4.1 FILE AND MEMORY REQUIREMENTS

The AlphaPascal system consists of these files:

```
DSK0:PRUN.PRG[1,4]
DSK0:CMPILR.PCF[7,5]
DSK0:PLINK.PCF[7,5]
DSK0:STDLIB.PCF[7,5]
DSK0:DEMO.PAS[7,5]
DSK0:DEMO.PCF[7,5]

DSK0:ERT.INC[7,5]
DSK0:SPOOL.INC[7,5]
DSK0:XLOCK.INC[7,5]
DSK0:XLOCK.SYS[1,4]
DSK0:XMOUNT.INC[7,5]

DSK0:PC.DO[2,2]
DSK0:PCL.DO[2,2]
DSK0:PL.DO[2,2]
DSK0:PCU.DO[2,2]
DSK0:PU.DO[2,2]
```

The first four of these files must be on your system if you are to use the AlphaPascal system. PRUN.PRG[1,4] is a re-entrant assembly language program; you may load it into system memory. CMPILR[7,5], PLINK[7,5], and STDLIB[7,5] are Pascal code file programs. (.PCF files may not be loaded into system memory.) DEMO.PAS and DEMO.PCF are the source and compiled versions of a sample Pascal program that demonstrates file handling. (This program also appears at the end of Chapter 10 of this book.)

The .INC files are special files you will include in programs that make use of several of the subroutines we have provided with the AlphaPascal system. (The special routines that make use of the .INC files are described in Chapter 14, "Systems Functions and Procedures.") (See Section 4.3.2.2, "The Include Option ($I)," for information on include files.)

The last five files listed above are .DO files: these are special command files that help you to compile and link files. They invoke the compiler and linker for you, and automatically answer all of the questions asked by those programs. Although these command files are not for use in all cases, you will probably be able to use them most of the time when you are compiling, linking, or updating a single file. For information on how to use these files, see Section 4.6, "Helpful Command Files."

4.1.1 File Extensions

Some of the extensions recognized by various components of the Pascal programming system are:

.PAS	Pascal source file, created by text editor.
.P01 .P02 .P03	Pascal intermediate files, created by the compiler. Not directly executable.
.PCF	Pascal code file. The executable program file created by the linker.
.PSB	Pascal assembly language subroutine.
.INC	Include files

NOTE: No .PSB files have been included with this release, although many of the routines in the standard library are actually linked-in assembly language programs. If you write your own assembly language subroutines, they must have the .PSB extension. The advantage in using assembly language programs in combination with your Pascal functions and procedures is that some systems functions can best be performed by an assembly language program because of speed, size, or hardware requirements.

4.1.2 File Search Pattern

Pascal uses a standard search pattern in looking for those files that it needs. For .PCF and .INC files, this pattern is:

```
The account you are logged into
Your project library account: [*,0]
The Pascal Library Account, PAS: -- DSK0:[7,5]
```

For PRUN.PRG, this pattern is:

```
System memory
User memory partition
System Library Account, SYS:-- DSK0:[1,4]
Your project library account: [*,0]
The account you are logged into
```

For .PSB files, this pattern is:

```
System memory
User memory partition
The account you are logged into
Your project library account: [*,0]
The Pascal Library Account, PAS: -- DSK0:[7,5]
```

For example, if you are logged into DSK1:[100,3], and want to execute the program PRIME.PCF, you enter:

```
PRUN PRIME (RET)
```

(PRUN assumes a file extension of .PCF.) Pascal first looks for the file PRIME.PCF in the account you are logged into (in this case, DSK1:[100,3]); next it looks in your project library account, DSK1:[100,0]. Finally it looks in the Pascal Library Account, DSK0:[7,5]. If it doesn't find the file in any of these places, you see the error message:

?Cannot OPEN PRIME.PCF - file not found

Of course, if you give a complete file specification (including device and/or account specification) Pascal will look for the file on the device and account you have specified, without going through its search pattern. The standard, complete AMOS file specification consists of a device specification, a file name, a file extension, and an account specification. For example:

PRUN HWK1:PRIME.PCF[200,56] (RET)

4.1.3 Program Restrictions

AlphaPascal handles your programs via a virtual memory paging system. This means that there is no limit to the size of your programs. (NOTE: Only programs are paged, not data allocated by NEW.) However, there are minor limits on the size of components of those programs:

1. The object code version of any one procedure may not be larger than 2000 bytes.

2. You may not have more than 255 global procedures and functions in any one program or library.

3. Any global procedure or function cannot have more than 255 local procedures or functions.

4. Maximum nesting of program block declarations is 15.

5. Maximum nesting of procedures, WITH-DOs, and RECORD type descriptions is 12.

4.1.4 Memory Requirements

Because AlphaPascal uses a virtual memory paging system, there is no limit to the size of your programs. However, a certain amount of memory is required to use CMPILR, PLINK, and PRUN. Although the minimum size of your memory partition depends on the data space requirements of the Pascal program you want to use, you should have at least 16K of memory to run a small program. To compile and link a program, you should have at least 24K of memory.

Also, you should note that even though you may execute a program that is larger than your memory partition, the larger that memory partition is, the less paging must be done and, in general, the faster your programs will run. To help even more in speeding up program execution and in reducing the minimum memory partition size, remember that you may load PRUN.PRG[1,4] into system memory. Also, if the assembly language subroutines that you write are re-entrant, you may load them into system memory. If you should run out of room in memory while compiling a program, CMPILR displays the messages:

```
?Insufficient memory
```

or:

```
?Attempt to call ERRORTRAP while in ERRORTRAP
```

4.2 CREATING A PASCAL PROGRAM

To create a Pascal source program, use one of the system text editors, VUE or EDIT. If you are using a video display terminal, you will probably want to use the screen-oriented text editor VUE. For a full description of how to use VUE and a list of all of its commands, see the AlphaVUE User's Manual, (DWM-00100-15). Also, Section 2.3.1, "Building a Pascal Program," of this book contains a brief introduction to VUE.

4.3 THE ALPHA PASCAL COMPILER

The compiler reads the source program that you have created, and compiles it into three intermediate files that have the same name as the source program file and the extensions .P01, .P02, and .P03. These files are used by the linker to create the final, executable program file, which has a .PCF extension. (If .P01, .P02, and .P03 files already exist with the same name as the program, CMPILR deletes them before compiling the new source program.)

To use the compiler, at AMOS command level enter:

```
PRUN CMPILR(RET)
```

The compiler now asks you for the name of the source file:

```
AlphaPascal V2.0
Source file name?
```

Enter the name of the file that contains the program or module you want to compile followed by a RETURN. (CMPILR assumes the .PAS extension.) This source file may be in any account, but the .P01, .P02, and .P03 files for the program will be generated in the device and account you are logged into.

4.3.1 The Diagnostic Display

After you have given CMPILR the name of the source file you want to compile, it asks:

```
Diagnostic file name (<return> for terminal)?
```

The diagnostic file contains information about the program compilation. You will usually want to see this information on the screen as the compilation proceeds, and therefore will enter a RETURN. If you want this information sent to a file so that you can have a permanent record of the compilation, enter a valid AMOS file specification. For example:

```
Diagnostic file name (<return> for terminal)? DIAG(RET)
```

The default extension is .LST. The diagnostic display might look something like this, depending on the program you are compiling:

```
AlphaPascal Compiler Version 2.0
          <   0>------
  NEWCHECK <   6>----
 PROGRAM   <  10>------
 16 lines
 7.07 seconds, 152.83 lines/minute
 No compilation errors.
```

The diagnostic display above shows the line numbers at which the procedures within the program begin (line #6 for the procedure NEWCHECK; line #10 for the main program). Each dash indicates the compilation of one program line. The last three lines tell you a) how many lines were in the program; b) how quickly the compilation was done; and c) how many errors occurred.

If an error occurs, you see it reported at the appropriate place in the compilation. For example, suppose we had left off a statement separator (the semicolon) at the end of the first line of the program. The diagnostic display would look like this:

```
AlphaPascal Compiler Version 2.0
          <   0>-
PROGRAM MYPROG
VAR     Target : REAL;
^
?Line 1: [INISOP] ';' or '(' expected -- inserting ';'
<   1>-----
  NEWCHECK <   6>----
 PROGRAM   <  10>------
 16 lines
 6.97 seconds, 155.02 lines/minutes
?Total of 1 compilation errors.
```

NOTE: If you tell CMPILR to send the diagnostic display to the terminal screen instead of a file, CMPILR pauses when an error occurs, and gives you a chance either to continue or quit. For example:

```
AlphaPascal Compiler Version 2.0
          <   0>-
PROGRAM MYPROG
VAR     Target : REAL;
        ^
?Line 1: [INISOP] ';' or '(' expected -- inserting ';'
Hit RETURN to continue
```

At this point you may continue the compilation by typing a RETURN, or you may stop the compilation by typing a Control-C (in which case you see the message: ?Compilation aborted). If an error occurs, CMPILR does not generate the .P01, .P02, and .P03 intermediate files; this is to prevent you from linking a program that contains a compile-time error.

4.3.2 Compiler Options

The AlphaPascal compiler has a number of options available to you. You may select one or more of these options at compile-time by including the appropriate option codes in your program.

You tell the compiler that you want to make an option request by including the symbol $ at the front of a program comment followed by the specific option code you want to use. The compiler acts upon the option requests as it reaches them in the program.

Option codes may be in upper or lower case. No space may separate the left comment delimiter and the option code. For example, {$G-} is valid, but { $G-} is not.

4.3.2.1 The GOTO Options ($G+ and $G-) - The $G+ code tells the compiler to allow use of the GOTO statement; the $G- code tells the compiler to generate an error message if it encounters a GOTO statement. You may use these options to turn GOTO recognition on and off within your program. (The compiler uses the $G- option as the default; that is, it does not recognize GOTO statements unless you use the $G+ option in your program.)

4.3.2.2 The Include Option ($I) - The $I code tells the compiler to include the contents of the specified file in your program. Supply a valid AMOS file specification. For example:

```
{$I MACRO.INC}
```

The default extension is .INC. The $I option code tells the compiler to physically insert the contents of the specified file into the file being compiled. The insertion takes place at the point of the option request. You may not include any other option codes after the file specification. The

purpose of the $I option is to save you from having to duplicate frequently used declarations or lines of code.

The include file can contain any valid program elements, as long as those elements can legally be inserted at the place in the program where the include file option occurs. (For example, you will not use the $I option request in a program's variable declaration section to include a file that contains a program header.)

NOTE: You cannot nest include file requests. That is, the include file may not itself contain an include file request.

4.3.2.3 The List Options ($L, $L+ and $L-) - The $L option request tells the compiler to send a listing to an AMOS file. (You do not see a program listing if you do not use the $L option.) Supply a valid AMOS file specification. For example:

```
{$L DSK1:DIAG[33,2]}
```

The listing will now be written to the specified file. The default file extension is .LST. If you do not give a file specification when you use the $L request, CMPILR creates a listing file bearing the name of your source file and a .LST extension in the account you are logged into.

Of course, you may not create a listing file outside of the project of the account you are logged into. For example, if you are logged into DSK0:[100,2] and try to create the listing file DSK0:LIST.LST[200,2], the AMOS system will respond with a "protection violation" error and abort the compilation because you tried to create the file in an account outside of the 100 project area.

You may use the codes $L- and $L+ to turn program listing off and back on again. For example, suppose you have a long program that contains a large section of comment that you don't want in your listing file. At the front of your source program you might say:

```
{$L MYPROG}
```

Directly in front of the section you do not want in your listing, you would place:

```
{$L-}
```

At the point where you want the listing turned back on again, place:

```
{$L+}
```

The compiler tells you in the diagnostic display that it is writing a listing file. For example:

```
AlphaPascal Compiler Version 2.0
        <   0>
List to LIST
-------
   ERRCHECK <   9>----
  PROGRAM  <  13>------
  20 lines
  12.36 seconds, 96.90 lines/minutes
 No compilation errors.
```

If you want the listing to appear on your terminal screen, use the device specification TTY:. For example:

```
{$L TTY:}
```

(NOTE: This display will be intermingled with that of the diagnostic display unless you send the diagnostic display to a file-- see Section 4.3.1, "The Diagnostic Display.") No other option requests may appear after the $L option. The listing consists of a display of your program with additional information to the left of the program. If your program contains errors, the listing file contains the appropriate error messages at the places in the program where the errors occurred. The listing takes a form that looks something like this, depending on the program you are compiling:

```
line# proc lv il dsp ic/lc
    1 --   D 1  0  1     1   {$L DIAG}
    2 --   D 1  0  1     1   PROGRAM Validate { Validate numeric entry; make
    3 --   D 1  0  1     1                      sure that it is between 1 and 100.};
    4 --   D 1  0  1     1
    5 --   D 1  0  1     1   VAR          Target : REAL;
    6 --   D 1  0  1     1
    7 --   D 2  0  2     4   FUNCTION ErrCheck(Local : REAL) : BOOLEAN;
    8 --   D 2  0  2     7    { Function checks entry. If 100<number<1,
    9 --   D 2  0  2     7      ErrCheck reports error by returning a TRUE. }
   10 --   C 2  1  2     0     BEGIN { Begin function ErrCheck }
   11 --   C 2  1  2     0       ErrCheck := Local < 1 OR Local > 100
   12 --   C 2  0  2    12     END { End function ErrCheck };
   13 --   C 2  0  2    40
   14 --   C 1  1  1     0   BEGIN { Main Program }
   15 --   C 1  1  1     2       WRITE('Enter a number between 1 and 100: ');
   16 --   C 1  1  1    45       READLN(Target);
   17 --   C 1  1  1    63       IF ERRCHECK(Target)
   18 --   C 1  2  1    69          THEN WRITELN('Invalid entry: try again.')
   19 --   C 1  3  1   112          ELSE WRITELN('Very good. Correct entry.')
   20 --   C 1  0  1   157   END { Main Program }.
0 compilation errors.
```

On the right you see a listing of the program. The left contains additional information about the program:

Line# - This is the number of the program line on the right-hand side of the display. The rest of the information on this line refers to this program line.

(Changed 30 April 1981)

Proc - You see the name of each locally declared procedure as CMPILR comes to it.

C or D - Pascal tells you if data (D) or code (C) is being generated for the program line.

lv and dsp - Internal information used by the compiler.

il - Indentation level. Tells you what nesting level the current program line is at.

ic/lc - Internal code location counter. This number tells you how many total bytes have been allocated at this point in the program compilation for the object code of the current procedure or function. The ic/lc number can come in handy later when you debug programs. If you interrupt program execution and backtrace that program, the backtrace gives you the "IPC" number-- the "Interpreter Program Counter." The IPC is the number designated by ic/lc in the program listing. You can thus compare your backtrace with your program listing, and see exactly where the problem occurs.

Also, if a run-time error occurs, the error message gives the IPC in the procedure at which the error occurred (e.g., ?Value range error in PROGRAM at IPC = 64 within FILL.PAS). (For information on backtracing, see Section 4.5.2, "Interrupting a Program.")

4.3.2.4 The Page Option ($P) - The $P option allows you to start a new page in the listing by telling the compiler to insert a form-feed at that point in the program listing. ($P is ignored if the $L option is not in effect.)

4.3.2.5 The Quiet Options ($Q+ and $Q-) - $Q+ designates the quiet-compile option. This option request tells the compiler to give you a brief diagnostic display, leaving off procedure names and line numbers. To turn full-display mode back on (the default condition), use the $Q- option code.

4.3.2.6 The Range Check Options ($R- and $R+) - The $R- option tells the compiler to turn off range checking; that is, the compiler does not output additional code to perform checking on array subscripts and assignments to subrange type variables. Programs compiled with range checking off run slightly faster; however, since the compiler is not checking for range errors, if an invalid index or assignment is made by your program, the run-time package will not stop the program when that error occurs. You

should not turn off range checking until your program has been tested and you are absolutely sure that your program runs without error. To turn range checking back on (the default condition), use the $R+ option.

4.4 THE ALPHA PASCAL LINKER

The linker, PLINK, reads the .P01, .P02, and .P03 files created by the compiler and resolves the files into a single executable program. You may use PLINK to link multiple files together into one program. However, even if your complete program consists of only one file, you must use PLINK on that file to generate an executable program file. The final file created by PLINK has the .PCF (Pascal Code File) extension. Although the use of PLINK may at first look complicated, once you begin to use it, you will find that its questions are rather self-explanatory. The paragraphs below discuss the different ways in which you can use PLINK. The last few paragraphs of this section (Sections 4.4.4 and 4.4.5) discuss the PLINK options.

To use PLINK, at AMOS commmand level enter:

PRUN PLINK (RET)

Now PLINK asks:

Code file:

Enter the specification you want given to your final .PCF file. This specification may be that of an existing file, or it may designate a new file. It may be the same as or different than the specification of one of the files you are going to link. Make sure that you supply a valid AMOS file specification that contains a filename of no more than six characters (for example, DSK4:VALID[110,4]). For the purposes of our discussions, let's say that you enter VALID

Code file: VALID (RET)

(PLINK will automatically assign the file a .PCF extension.) If you do not include a device and account specification, PLINK assumes that you want to link a file that is in the device and account you are logged into. At this point PLINK asks you different questions, depending on whether or not the specified .PCF file already exists. In the next sections we will step through the three situations which can occur: 1) you are creating a new file; 2) you are replacing an existing .PCF file; 3) you are updating a single module in the .PCF file.

For now, let's assume that PLINK has asked its next few questions, and knows what files to link together and what external library to use. You see:

Loading program and library dictionaries

This tells you that PLINK is getting ready to process your file. For each file that you are linking, PLINK tells you when it begins working on that file. For example:

```
Processing NEWMOD
```

Next PLINK tells you what globally declared functions and procedures are being linked into your .PCF file. (These routines are in your program and the external library.) For example:

```
Linking in global func/proc ERRCHECK
Linking in global func/proc PROGRAM
```

At last, PLINK is finished, and begins to copy the resolved code into the .PCF file:

```
Transferring temporary file to new code file
```

PLINK's final message tells you that it is finished:

```
VALID completed
```

Now, let's get back to the questions PLINK asks when it is determining which files to link together. NOTE: Keep in mind when answering PLINK's questions that PLINK converts all of your input to upper case.

4.4.1 Linking a New .PCF File

If you use PLINK to create a .PCF file, and that file does not already exist, PLINK knows that you are linking a new program, and not trying to replace or update an existing program. For example, suppose you have told it that you want to create VALID.PCF. It tells you:

```
Creating new code file VALID.PCF
```

Now it asks which external library you want to use for the new program:

```
Library code file for VALID.PCF =
```

Enter the file specification of the library you want to use. In almost every case, this will be the standard library file, STDLIB.PCF. The external library contains routines used by your program and the compiler. You must specify a library (except in the very rare case where you are linking a "root" library-- that is, a library that has no library of its own-- such as STDLIB itself). For information on the external library, see Chapter 16, "Writing and Modifying an External Library File." Now PLINK asks which files you want to link together:

```
Please specify files to be linked into VALID,
  one per line, ending in a blank line

  File 1 =
```

Enter the specification of the first file; then type a RETURN. Now PLINK asks for another file:

```
  File 2 =
```

Remember that a single .PCF file may be made up of several separately compiled modules. If you are linking only one file, enter a RETURN here; otherwise enter the file specification of the next module. If you are linking together more than one file, the file specifications do not have to be entered in any special order, but at least one of these files must be a main program file (rather than a module), or you see the message: ?Attempt to create new code file without main program block. (For information on module files, see Section 6.1, "Program Declarations.") Remember that you are entering AMOS file specifications, and not the internal names of your programs or modules; each specification must contain a six-character or less file name that designates an AMOS disk file.

NOTE: Although you will usually be linking together compiled Pascal files, you may also want to use .PSB (Pascal assembly language subroutine) files. To tell AlphaPascal that a file is an assembly language subroutine rather than a Pascal program file, you will specify the .PSB extension. For example:

```
File 1 = MODUL1 (RET)
File 2 = MAINPR (RET)
File 2 = INPUT.PSB (RET)
File 3 = ANYCN.PSB/LINK (RET)
File 4 = (RET)
```

The example above shows us linking together a main program file, MAINPR, a module file, MODUL1, an assembly language subroutine file reference, INPUT.PSB, and an assembly language subroutine, ANYCN.PSB. For a discussion of how these .PSB files are linked in, see Section 4.4.4, "Linking Assembly Language Subroutines (the /LINK Option)." For information on assembly language subroutines, see Chapter 15, "Assembly Language Subroutines."

4.4.2 Replacing a .PCF File

If the VALID.PCF file that we specified as "code file" already exists, PLINK knows that we want to either update or replace the file. Therefore, after it asks for the code file, PLINK asks:

```
Do you wish to 1) replace or 2) update VALID.PCF?
```

To replace the file, enter a 1. PLINK now says:

```
Creating new code file VALID.PCF
```

It asks which external library to use:

```
Library code file for VALID.PCF =
```

Once again, you will probably want to answer "STDLIB." Now PLINK asks for the names of the files you want to link together:

```
Please specify files to be linked into VALID,
  one per line, ending in a blank line

  File 1 =
```

Enter the specification of the first file; then type a RETURN. Now PLINK asks for another file:

```
  File 2 =
```

Type a RETURN if you are only linking one file; otherwise, supply the file specification of the next module. When you have finished entering all module specifications, enter a single RETURN. (See Section 4.4.4 for information on linking assembly language subroutines.)

4.4.3 Updating a .PCF File

It would be extremely inconvenient to re-compile and re-link a huge Pascal program every time you wanted to change a tiny portion of it. AlphaPascal allows you to split one program up into a number of files called "modules," which are linked together with one main program file. You can change a module file, re-compile just that file, and then re-link the changed module into the main .PCF file.

To update a single module, make your changes and then re-compile that module. Now, use PLINK to re-link the module into the program. When PLINK says:

```
Do you wish to 1) replace or 2) update VALID.PCF?
```

enter a 2 followed by a RETURN. Now it will tell you what external library was used to link that .PCF file. For example:

```
The standard library code file for VALID.PCF is STDLIB.PCF
Do you wish to change this?
```

Answer Y or N. You will probably want to answer N, to instruct PLINK to use the same library the file was originally linked with. If you answer Y, PLINK asks for the new library:

```
New standard library =
```

Enter the specification of the external library you want to use.

Now PLINK asks what files you want to link together. Just enter the specifications of the module or modules you have re-compiled. The rest of the modules in the .PCF file will be left alone. NOTE: If you do change to a new library, you will have to re-link all modules used in the program and the main program file, since the old modules will be incompatible with the new library. (See Section 4.5.1, "Library Version Checking," for more information on program-library compatibility.)

PLINK will tell you what new procedures or functions have been linked in, and what old procedures or functions have been kept. For example:

```
Keeping global func/proc ERRCHECK
Keeping global func/proc PROGRAM
Linking global func/proc NEWPROC
```

4.4.4 Linking Assembly Language Subroutines (the /LINK Option)

We mentioned briefly above in Section 4.4.1, "Linking a New .PCF File," that you can link assembly language subroutine (.PSB) files into your .PCF file by specifying the .PSB extension when you use PLINK to link the subroutine files into the program. (For information on such routines, see Chapter 15, "Assembly Language Subroutines.")

What actually happens is this: when you specify a .PSB file to PLINK, PLINK then inserts a reference to that file in your final .PCF file. When you execute the .PCF file, AlphaPascal searches for the specified .PSB file (using the standard file search pattern we discussed at the front of this chapter), and then loads that file into memory from the disk (if the file is not already in system or user memory); next, it executes the routine when called by the program. When PRUN finishes executing the .PSB file, it deletes it from memory. (You can force PRUN to leave the .PSB file in memory by explicitly loading the file into memory via the monitor LOAD command before using PRUN to run the program that calls the .PSB file. If the .PSB file has been placed into memory via the LOAD command, the file remains in memory until you use the DEL command to remove it.)

If you want the contents of the .PSB file to be physically part of your .PCF file (so that this search-and-load procedure does not take place), you may specify the /LINK option after the name of the .PSB file when you link that file in. For example:

```
File 1 = MODUL1 (RET)
File 2 = MAINPR (RET)
File 3 = XPUT.PSB/LINK (RET)
```

The /LINK option refers only to the single file specification on the same line as the option request. If you are going to physically link a .PSB file into your .PCF file, the .PSB file cannot be larger than one disk block.

NOTE: Usually if you modify a module or .PSB file, you only need to re-link the modified file into the linked .PCF file of which it is a part. (For example, if you changed the file XPUT.PSB in our example above, you would not need to re-link MAINPR and MODUL1; only XPUT.PSB.) However, if you decide to replace a .PSB file with a Pascal file of the same name or vice versa, you will need to re-link all modules that form the .PCF file of which that file is a part. For example, looking at our example above again, if you decide that the file MODUL1 would be better as an assembly language file, MODUL1.PSB, you will need to re-link all of the files that form the complete .PCF file--MODUL1.PSB, MAINPR, and XPUT.PSB.

4.4.5 Preventing Backtracing of .PCF Files (the /SMASH Option)

AlphaPascal allows you to trace the functions and procedures called by a program. This is a useful debugging feature when you are developing a program, since you can interrupt the program at a trouble spot and see what function or procedure it is in. (For more information on backtracing, see Section 4.5.2, "Interrupting a Program.")

However, once a program has been finished and tested, you may not want users of that program to be able to find out the names of the program functions and procedures (which they can ordinarily do by interrupting the execution of the program and backtracing). Therefore, AlphaPascal provides the linker /SMASH option.

When you link a program using the /SMASH option, users of that program are prevented from seeing the names of the program's procedures and functions when they backtrace the program; instead, the names are replaced with asterisks. For example, instead of the backtrace display:

```
Interrupt (?=Help): B (RET)

    In STDLIB.PCF
      RDR       at IPC = 33
    In VALID.PCF
      PROGRAM   at IPC = 43
    In STDLIB.PCF
      PROGRAM   at IPC = 423
    Exit to AMOS
```

they see:

```
Interrupt (?=Help): B (RET)

  In STDLIB.PCF
    RDR       at IPC = 33
  In VALID.PCF
    ******** at IPC = 43
  In STDLIB.PCF
    PROGRAM   at IPC = 423
  Exit to AMOS
```

Note that in the smashed version above, the name of the function in your own program, VALID.PCF, is blanked out with a line of asterisks.

To use the /SMASH option, place the option request after the name of the code file you want to smash. For example:

```
Code file = VALID/SMASH (RET)
```

When PLINK finishes linking the specified files, it tells you that the names of the functions and procedures in the code file have successfully been hidden from the backtrace option. In the case of the file discussed above, VALID.PCF, you see:

```
SMASHed VALID/SMASH Completed
```

NOTE: CMPILR and PLINK have both been linked using the /SMASH option.

4.5 THE ALPHA PASCAL RUN-TIME PACKAGE

The AlphaPascal run-time package, PRUN, is the program that executes your program by interpreting the .PCF file created by the linker. To use PRUN, at AMOS command level enter PRUN followed by the specification of the file that contains the program you want to execute. Then type a RETURN. For example:

```
PRUN LSTSQR[200,1] (RET)
```

4.5.1 Library Version Checking

Because you can add routines to the external library, the situation can arise where an old program was linked with an external library that is different from the current external library. PRUN will not execute a program that is not compatible with the library it is being run with. By "compatible," we mean that a program that was linked with a certain external library cannot be run with an older version of that library, or with a completely different library.

You will rarely have to worry about library version numbers; if you modify a library, you can run programs linked with earlier versions of that library without re-linking the programs (unless you changed functions and procedures used by those programs, in which case you might have to change your programs to be compatible with the new procedures and functions).

AlphaPascal uses a system of version numbers and version stamps to keep track of program and library versions. (These numbers are for internal use only-- they are not accessible to your programs.) Whenever a library is created or modified, AlphaPascal writes a unique identifying number called the "version stamp" to that library. It also keeps track of the number of version stamps generated for a library; this number is called the "version number."

Whenever you link a program, AlphaPascal writes the version stamp and version number of the external library you are using to the .PCF file being linked. Whenever you execute a program, PRUN checks to make sure that the version stamp for that program matches one of the program stamps in the current external library. This makes sure that the current library is not a completely different library than the one the program was linked with. If the library is a modified version of the library the program was linked with, checking to see that the version stamp in the program exists in the list of version stamps in the library makes sure that the library is not an earlier version than the library with which the program was linked.

If the library and program are not compatible, you cannot run the program with that version of the library; instead, you must re-link your program with the current library.

PRUN displays the following message if the program version stamp and number of the library are older than those of the program:

```
?Wrong version of xxxx for use with yyyy
```

where xxxx is the external library, and yyyy is your .PCF file.

If you update an external library, check to see if your old .PAS files have to change because of the revisions. For example, if a hypothetical procedure REVERSE now expects three arguments, while a previous version expected two, your programs will have to change to accommodate the changes in the procedure. (For more information on the external library, see Chapter 16, "Writing and Modifying an External Library File.")

4.5.2 Interrupting a Program

Whenever you use PRUN, you can tell it to interrupt program execution by typing a Control-C. PRUN stops the program being executed and displays:

```
Interrupt (?=Help):
```

You may enter one of four responses: Q, R, B, or ?, followed by a RETURN:

Q - Tells PRUN that you want to terminate program execution. PRUN returns you to AMOS command level.

R - Tells PRUN to resume program execution at the point of interruption.

B - Tells PRUN to print a backtrace of all the procedures and functions invoked during the program execution to this point. These procedures and functions are listed in order, with the last-called procedure or function listed first. The display might look something like this, depending on the program you are executing:

```
Interrupt (?=Help): B (RET)

    In STDLIB.PCF
      RDR       at IPC = 33
    In VALID.PCF
      PROGRAM   at IPC = 43
    In STDLIB.PCF
      PROGRAM   at IPC = 423
    Exit to AMOS

Interrupt (?=Help): Q (RET)
.
```

(For information on keeping program users from using the backtrace function to see the names of the functions and procedures in your programs, see Section 4.4.5, "Preventing Backtracing of .PCF Files (the /SMASH Option).")

? - Tells PRUN that you need help. PRUN now displays a menu of the responses you can enter:

```
Interrupt (?=Help): ? (RET)

    Q = Quit
    B = Backtrace
    R = Resume

Interrupt (?=Help):
```

4.6 HELPFUL COMMAND FILES

Although our discussions above on the compiler and linker discussed several special uses of those programs, in general the information that you give to the programs will be fairly standard. For example, you will rarely want to use an external library other than STDLIB. To make CMPILR and PLINK easier to use, we have provided a number of special command files that you can use for most cases of compilation and linking; these files automatically supply much of the information needed by CMPILR and PLINK.

These command files are in the Command File Library Account, DSK0:[2,2]. (A command file is a text file that contains a series of AMOS commands and input for those command programs. Such a file allows you to execute a string of commands and provide a stream of input by simply entering the name of that file.)

You will use these command files at AMOS command level. To invoke one of the files, enter the name of the file followed by one or more file specifications. For example, suppose you want to use the command file named PC (for Pascal-compile) to compile your program file SMALL.PAS. At AMOS command level, enter:

```
PC SMALL (RET)
```

The PC command file now invokes the Pascal compiler, and tells it that you want to compile the file SMALL. Then it tells CMPILR that you want the diagnostic file to be displayed on the screen. NOTE: If an error occurs while you are using one of these command files (for example, if your program contains an error or if AMOS cannot find the specified file), AlphaPascal stops execution of the command file. After you clear up the problem, you can then use the command file again.

The command files we have provided are:

DSK0:PC.DO[2,2]	Pascal-compile
DSK0:PL.DO[2,2]	Pascal-link
DSK0:PCL.DO[2,2]	Pascal-compile and -link
DSK0:PCU.DO[2,2]	Pascal-compile and -update
DSK0:PU.DO[2,2]	Pascal-update

Remember that these command files do not cover all cases of compiling and linking files. If after you read these descriptions you realize that the file will not do exactly what you need, you will have to run CMPILR and PLINK yourself to perform the actions you want.

4.6.1 Compiling a Single File (PC.DO)

To use the PC file, enter PC followed by the name of the file that contains the program you want to compile. Then type a RETURN. For example:

```
PC DRWLIN (RET)
```

CMPILR compiles the file DRWLIN.PAS into the files DRWLIN.P01, DRWLIN.P02, and DRWLIN.P03.

4.6.2 Linking a Single File (PL.DO)

To use the PL file, enter PL followed by the name of the file that you want to link. For example:

PL DRWLIN (RET)

PLINK now links the files DRWLIN.P01, DRWLIN.P02, and DRWLIN.P03 together into DRWLIN.PCF. Before you try to link a file, make sure that it has already been compiled; that is, that the .P01, .P02, and .P03 files exist. PL assumes that you want to link a single file, and that you want to use the standard external library, STDLIB.

4.6.3 Compiling and Linking a Single File (PCL.DO)

To use the PCL file, enter PCL followed by the name of the file you want to compile and link. For example:

PCL TRSRCH (RET)

The compiler compiles the file TRSRCH.PAS into the files TRSRCH.P01, TRSRCH.P02, and TRSRCH.P03. Next, PLINK links these intermediate files into TRSRCH.PCF. The command file assumes that you want to link a single program file, and that you want to use the standard external library, STDLIB.

4.6.4 Updating a Single Program Module (PU.DO)

To use the PU file, enter PU followed by the name of the module you want to update, followed by the name of the .PCF file you want to link the module into. For example:

PU MODUL1 TRSRCH (RET)

PLINK now links the module into TRSRCH.PCF. This file assumes that MODUL1 has already been compiled, and that you want to use whatever external library TRSRCH was originally linked under.

4.6.5 Compiling and Updating a Single Program Module (PCU.DO)

To use the PCU file, enter PCU followed by the name of the module you want to compile and update, followed by the name of the .PCF file you want to link the module into. For example:

```
PCU MODUL1 TRSRCH (RET)
```

CMPILR now compiles MODUL1; PLINK then links it into TRSRCH.PCF. PCU assumes that you want to use the external library TRSRCH was originally linked under.

PART II

SUMMARY OF ALPHA PASCAL

The next nine chapters discuss the elements of the Pascal language as it has been implemented by Alpha Micro. If you are interested in a quick summary, refer to Appendix A, "A Quick Reference to AlphaPascal."

CHAPTER 5

GENERAL INFORMATION

This chapter contains very general information about AlphaPascal program concepts such as: basic program structure, statement separation and spacing, legal identifiers, compound statements, scope of identifiers, etc. For detailed information on specific elements of a Pascal program, see the Index and Appendix A, "Quick Reference to AlphaPascal."

5.1 BASIC STRUCTURE OF A PROGRAM

This section lists the major elements of a Pascal program. We'll talk more about each element in the following paragraphs, but this will give you a general idea of what goes where. Every Pascal program follows the general form:

```
Heading
    block.
```

The heading follows this form:

```
PROGRAM program-name;
```

or:

```
PROGRAM;
```

NOTE: Standard Pascal requires that you follow the program-name with a set of names that are concerned with program input and output (for example: PROGRAM Schedule(INPUT,OUTPUT);). AlphaPascal, however, ignores these names, and you may omit them altogether. However, make sure that the program heading is syntactically correct. (For example, "PROGRAM NewAccount (;" generates an error message because of the open parenthesis.)

The program block which appears under the heading consists of a declaration section that defines the names and properties of various data objects (such as variables and constants) and subprograms (such as procedures and functions) that will be used in the program, and a statement section, which

lists the actions to be taken upon the declared items. (The names of the data objects, as well as the names of the procedures and functions of a Pascal program are called "identifiers.") The program block takes this form:

```
Label-declaration part
Constant-definition part
Type-declaration part
Variable-declaration part
External-declaration part
Procedure-and-function-declaration part
Statement part.
```

(For information on the definition and declaration sections of the program block, see Chapter 6, "Declarations and Definitions." For information on the statement section of the program block, see Chapters 9-13.

Any number of spaces and/or blank lines may appear between words and symbols in a Pascal program. Because program statements may be broken up by blank lines and spaces, Pascal requires that you identify where one statement ends and another begins by separating them with a semicolon. For example:

```
PROGRAM NewTest;

VAR Counter : REAL;
```

The last element of a Pascal program must be the END keyword followed by a period. (The period indicates that the end of the program has been reached, rather than just the end of a group of statements within the program.)

As a final word on program structure, we would like to mention that your program can consist of more than one file. The advantage of splitting your program up into multiple files is that when a change needs to be made to one of the files, you only have to re-compile the one file and then re-link the files, rather than re-compile all of the files.

If your program does consist of multiple files, only one of those files will follow the main program format we discussed above; the rest will follow a slightly different format. (This is because only one main program file may be linked together with other files.) These non-program files follow this format:

```
MODULE module-name;
     block.
```

or:

```
MODULE;
     block.
```

This heading tells the Pascal compiler that the file is not a main program file, and that it is part of a multiple-file program. The module-name identifies this non-program file, and does not necessarily have to be the

same name as that assigned to the actual file or to the main program.

The block takes this form:

```
Label-declaration part
Constant-definition part
Type-declaration part
Variable-declaration part
External-declaration part
Procedure-and-function-declaration part
.
```

As you can see, the file does not contain a statement part. (Although, of course, the procedure and function declarations can contain a statement section.) The file ends with a period (even though it cannot end with an END keyword followed by a period).

Below is a small sample of a module and the main program with which it is linked:

A MODULE

```
MODULE;

FUNCTION MAX(Arg1,Arg2 : REAL) : REAL;
  BEGIN { MAX }
    IF Arg1>Arg2 THEN MAX:=Arg1 ELSE MAX:=Arg2
  END { End of MAX };
.
```

A MAIN PROGRAM

```
PROGRAM Main;

VAR   Num1, Num2 : REAL;

EXTERNAL FUNCTION MAX(Arg1,Arg2 : REAL) : REAL;

BEGIN { Main Program }
  WRITE('Enter two numbers: '); READLN(Num1,Num2);
  WRITELN;
  WRITELN('The larger number is:',MAX(Num1,Num2))
END { Main Program }.
```

5.2 COMPOUND STATEMENTS (BEGIN AND END)

The statement section of the program block starts with a BEGIN keyword and ends with an END keyword. The elements within these two keywords may consist of one statement or many, and comprise the executable section of the program.

Any one statement may be replaced by a combination of statements called a "compound statement." A compound statement is a series of statements, and starts with the BEGIN keyword and finishes with the END keyword. By convention, the programmer usually indents each compound statement one level within the program (see the example below) so that he or she can visually keep track of how compound statements are nested.

The individual statements within the compound statement must be separated by a semicolon. For example:

```
PROGRAM Average;
{ This program computes the average of a series of numbers }

VAR                    Count : INTEGER;
        Answer, Total, Num : REAL;
CONST Maxval = 10;

BEGIN { Average }
   Total := 0
   FOR Count := 1 TO Maxval DO
     BEGIN
       WRITE ('Enter number, please: ') { Prompt user for number };
       READLN(Num) { Get number from user };
       Total := Total + Num { Sum numbers }
     END;
   Answer := Total/Maxval { Compute average };
   WRITELN ('Average is: ', Answer)
END { Average }.
```

In the example above, the statement section of the program block contains two nested BEGIN-END compound statements. Note that the BEGIN keyword does not require a semicolon after it, and that you do not precede the END keyword with a semicolon. This is because BEGIN and END are keywords, but are not statements. Therefore, there is no need to separate BEGIN from the WRITE statement; in fact, doing so causes an error. For the same reason, do not place a semicolon between the END keyword and the statement before it.

5.3 COMMENTS

Sometimes the function of a section of a program is not immediately obvious to the casual observer. To help the reader of a program understand what that program is doing, Pascal allows you to enter "comments" in your program.

Comments are ignored by the compiler, and serve only to document the source program. AlphaPascal accepts as comments any text enclosed either by a pair of "{}" or "(* *)" symbols. For example:

```
READ(PlaneRoute) (* This variable is accessed by FLIGHT procedure *);
```

A comment may appear between any two symbols in a program, cover more than

one line, and may appear in the middle of a statement. Comments may not be nested, but {} symbols may appear within the symbols (* *), and vice versa; this allows you to "comment out" areas of programs that contain comments. For example:

```
(* WRITELN(RecCount)              { Report # of records sorted }
   READLN;
   IF Error THEN ErrorFix { Error condition }; *)
```

5.4 LEGAL IDENTIFIERS

Identifiers are groups of characters that denote variables, types, constants, procedures, functions, programs, record fields, and record tagfields. As one example of an identifier, consider a variable that assumes the values of a range of school test scores; it might appear in a program as the identifier Scores.

Identifiers in AlphaPascal may consist of combinations of upper and lower case letters and numbers, but must begin with a letter. Identifiers may be as many characters as you wish, but only the first eight characters are used by Pascal in recognizing the identifier. (That means that the identifiers STANDARDBUFFER and STANDARDBUFFOON will be recognized by Pascal as the same identifier-- STANDARD.)

IMPORTANT NOTE: AlphaPascal "folds" lower case identifiers to upper case. This means that it translates all lower case letters to upper case when considering identifiers. In other words, AlphaPascal considers the identifiers EvalQuote, Evalquote, EVALQUOTE, and evalQUOTE to be the same identifier.

You may choose any combinations of letters and numbers for identifiers with the following exceptions. Certain words (called "keywords" or "reserved words") have been reserved by Pascal to identify statements and structures inherent to Pascal, and may not be used as identifiers. (These keywords are listed in the section below.) Other identifiers (called "standard identifiers") have been pre-declared by AlphaPascal. This means that AlphaPascal recognizes these standard identifiers as denoting procedures, functions, and types already defined to AlphaPascal. The difference between standard identifiers and keywords is that you MAY redefine standard identifiers so that they no longer represent predefined Pascal types, functions, and procedures. In other words, if you attach a new meaning to a standard identifier, no error message is generated; but, the procedure, function or type previously associated with that identifier is no longer available to the procedure or function in which you redefined the identifier. (Of course, a re-definition only applies to the program in which it appears.)

For this reason, you must be very careful when assigning identifiers not to inadvertently redefine a standard identifier whose procedure, type, or function you may have need for later on in the program.

5.4.1 Reserved Words

Below is a list of the Pascal reserved words. You may not use these reserved words as identifiers.

AND	ARRAY	BEGIN	CASE	CONST
DIV	DO	DOWNTO	ELSE	END
EXTERNAL	FILE	FOR	FUNCTION	GOTO
IF	IN	LABEL	MOD	MODULE
NIL	NOT	OF	OR	PACKED
PROCEDURE	PROGRAM	RECORD	REPEAT	SET
THEN	TO	TYPE	UNTIL	VAR
WHILE	WITH			

5.4.2 Standard Identifiers

Below is a list of all AlphaPascal standard identifiers. You may redefine these identifiers. However, be careful not to unintentionally redefine them.

Constants:

FALSE	TRUE	MAXINT

Types:

INTEGER	BOOLEAN	REAL	CHAR	STRING
TEXT				

Predeclared files:

INPUT	OUTPUT	KEYBOARD

Procedures, variables and functions. (NOTE: Several of these procedures, variables, and functions are for internal use of the compiler and standard library. For a list of all functions and procedures available for your use, refer to Appendix A, "Quick Reference to AlphaPascal," or to the Table of Contents.)

ABS	ARCCOS	ARCCOSH	ARCSIN
ARCSINH	ARCTAN	ARCTANH	CHARMODE
CHR	CLOSE	CONCAT	COPY
COS	COSH	CREATE	CRT
DELETE	EOF	EOLN	ERASE
ERROR	ERRORINFO	ERRORTRAP	EXIT
EXP	EXPONENT	EXTENSION	FACTORIAL
FILLCHAR	FILESIZE	FSPEC	GET
GETFILE	GETLOCKS	IDSEARCH	INCHARMODE
INSERT	JOBDEV	JOBUSER	KILCMD
LCS	LENGTH	LINEMODE	LN
LOCATION	LOG	LOOKUP	MAINPROG
MARK	MEMAVAIL	MOVELEFT	MOVERIGHT
NEW	ODD	OPEN	OPENI
OPENO	OPENR	ORD	PAGE
PFILE	POS	POWER	PRED
PUT	PVIRT	PWROFTEN	PWROFTWO
RAD50	RANDOMIZE	RDC	RDI
RDR	RDS	READ	READLN
RELEASE	RENAME	RESET	REWRITE
RLN	RND	ROUND	SCAN
SEEK	SETFILE	SHIFT	SIN
SINH	SIZEOF	SPL	SPOOL
SQR	SQRT	STDERRORTRAP	STR
STRIP	SUCC	TAN	TANH
TIME	TOD	TREESEARCH	TRUNC
UCS	VAL	WLN	WRB
WRC	WRI	WRITE	WRITELN
WRR	WRS	XERRORTRAP	XLOCK
XMNT	XMOUNT		

5.5 SCOPE OF IDENTIFIERS

Because Pascal is a block structured language, a Pascal program falls naturally into a nested structure. (See Figure 5-1, below. Each Block in the diagram represents some procedure or function within the program.) What happens if, for example, a variable is declared in the main program, and then re-declared in a procedure called by the main program? Which declaration is valid? This problem is resolved by defining the "scope" of the identifier; that is, by defining the area of a program for which the declaration of an identifier is valid.

The scope of an identifier is the program, procedure, or function in which it is defined and any enclosed blocks which do not redefine it. (The use of an identifier in the same block as its declaration is called a "local" reference; the use of an identifier declared in an outer block is called a "non-local" reference.)

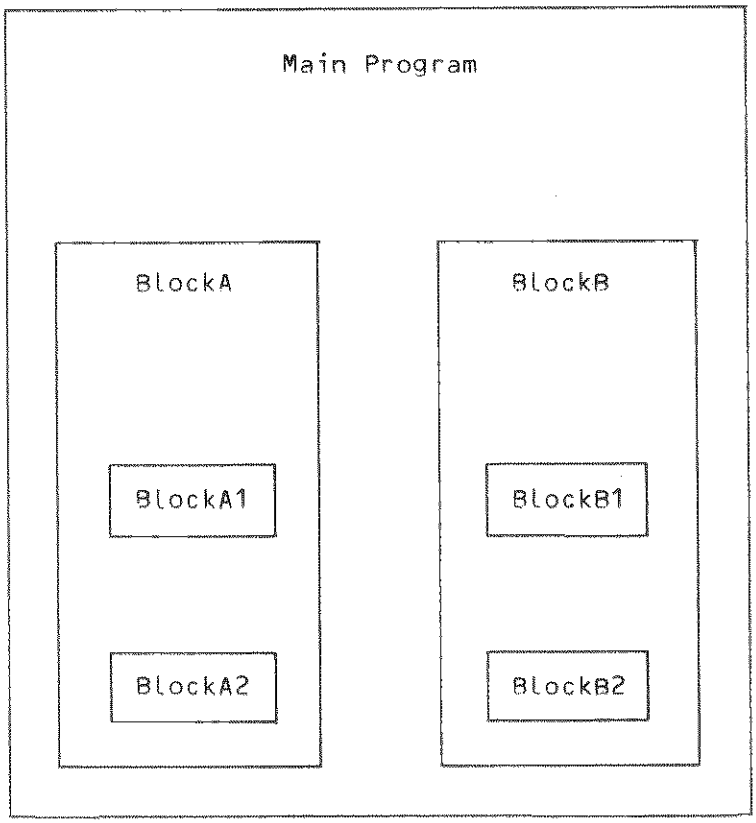


Figure 5-1

Nested Structure of Program Blocks

Let's say that a constant is defined both in the main program and BlockA. BlockA itself and the blocks enclosed in BlockA (BlockA1 and BlockA2) use the definition made in BlockA. The main program, BlockB, BlockB1, and BlockB2 use the constant definition made in the main program.

The following small program demonstrates identifier scoping. The variable Counter is declared both within the main program and within the procedure InnerBlock:

```
PROGRAM Scope { This program tests identifier scoping };

VAR   Counter : INTEGER;
  { "Counter" declared for main program }

PROCEDURE InnerBlock;
VAR   Counter : INTEGER;
  { "Counter" declared for Procedure InnerBlock }
  BEGIN
    Counter := 1;
    FOR Counter := 1 TO 10 DO
      BEGIN
        WRITELN('Procedure InnerBlock-- Counter = ',Counter);
      END
  END { End Procedure InnerBlock };

BEGIN { Main Program }
  Counter := 20;
  WRITELN('Main Program-- Counter = ',Counter);
  InnerBlock { Invoke Procedure InnerBlock };
  WRITELN('Main Program again-- Counter = ',Counter)
END { End Main Program }.
```

If our description of identifier scoping is correct, we would expect the statement:

```
WRITELN('Main Program again-- Counter= ',Counter)
```

to produce the value 20, regardless of the value assumed by Counter within the procedure InnerBlock. That is exactly what happens.

5.6 NOTATION

AlphaPascal uses several conventions in handling and representing numbers and strings.

5.6.1 NUMBERS

Pascal recognizes two types of numbers: integer and real. The integer numbers are the "whole numbers"; that is, they cannot contain a fractional part. Real numbers are numbers that contain a decimal point, and which therefore contain a fractional part (even if that fractional part is zero).

For example, these numbers are integers:

```
-231
   7
8098
```

These are real numbers:

```
567.8
-25.00
  4.318
```

(For information on the REAL and INTEGER data types, see Chapter 7, "Data Types.") Pascal has two methods of displaying numbers: decimal notation and scientific notation. Decimal notation allows us to represent a number with an optional sign, a whole number part, a decimal point, and an optional fractional part. If the fractional part exists, there must be at least one digit on each side of the decimal point. For example:

```
-2405.3
```

Scientific notation is handy for representing very small or very large numbers. A number represented in scientific notation is shown as a value multiplied by the appropriate power of 10. To indicate the exponent, Pascal uses the symbol "E". For example:

```
-2.4053E+3
```

represents "negative 2.4053 times 10 to the third"; that is, in decimal notation, the number would be -2405.3. A positive number after the E tells you how many places to shift the decimal point to the right, in order to read the number in decimal notation; a negative number tells you how many places to shift the decimal point to the left. For example, to represent the number:

```
5.678E-2
```

in decimal notation, shift the decimal point to the left two places: 0.0567.

AlphaPascal generally uses decimal notation to display real and integer numbers. (Of course, if the number is integer, no fractional part is shown.) However, if a number is too large or too small to represent easily in decimal notation, AlphaPascal displays it in scientific notation.

You may use either scientific or decimal notation when entering numbers to a Pascal program, or within the program itself.

(For information on using the WRITE and WRITELN procedures to format numeric and character output, see Section 10.1.5.5, "Formatting Output.")

5.6.2 STRINGS

A string is a group of characters. These characters may be numbers, letters, or any combination of characters, including the delimiters for a comment-- {} or (* *). A string is identified to Pascal by enclosing it in single quotation marks. For example:

```
'This is a string.'
'Data: 123'
'The END is near'
```

The characters in a string represent themselves, rather than numeric values, reserved words, etc. For example, the third example contains the characters "123", but does not represent the number 123. The fourth example contains the characters "END", but does not represent the keyword END.

If you wish a string to contain a quotation mark, place two quotation marks where you want the single quotation mark to appear. For example:

```
'You don''t say.'
```

A string may be defined in a constant definition. For example:

```
CONST Message = 'Error - Type CR to recover'
```

(We then say that Message is a string constant.) Or, a string may be used as a string literal. For example:

```
WRITELN('Do not forget to write-enable the disk.')
```

NOTE: AlphaPascal includes the data type STRING as a standard data type. Data of type STRING consists of a group of characters (data of type CHAR) rather than a single character. For information on CHAR and STRING, see Chapter 7, "Data Types."

CHAPTER 6

DECLARATIONS AND DEFINITIONS

One of the important features of Pascal is that it requires that you define and name the data objects you are going to use in a program before you reference those objects. For example, if you are going to be using a variable named "Cost", you must "declare" that variable at the start of the program or procedure in which that variable appears. Besides declaring variables, you must also declare the program name, labels, functions and procedures, and modules. In addition, you must define any numeric or string constants you are going to use, as well as any data types. All declarations and definitions appear at the front of the main program or the procedure or function containing the declared data objects.

These centralized declarations and definitions greatly enhance the legibility and organization of your program, and aid the compiler in performing error detection.

You'll remember from Chapter 5 that the declaration and definition part of the program block takes the form:

Label declarations
Constant definitions
Type declarations
Variable declarations
External declarations
Procedure and function definitions

6.1 PROGRAM DECLARATIONS

The program declaration consists of the PROGRAM keyword. It may also contain a program name. This program declaration assigns the name of the main program, and marks the start of the main program file. A program name may be any legal identifier (see Section 5.4, "Legal Identifiers"). The program declaration statement takes the form:

PROGRAM program-name;

or:

PROGRAM;

Unlike other versions of Pascal, AlphaPascal does not require or recognize any information about external input or output files after the program name in the program declaration. Neither does AlphaPascal attach any significance to the program name. That is, the program name serves only as a type of comment, and does not actually identify the file.

End the program declaration with a semicolon to separate it from the rest of the program statements. For example:

PROGRAM BubbleSort;

An AlphaPascal program may consist of more than one file. You can compile these files separately; then, using PLINK, you can link them together into one program. Of the files that you are going to link together, only one may be a main program file. You tell the linker which files are not the main program file by including an external program declaration at the front of those files. This declaration tells the linker that the file is not the main program (that it is, in effect, an external file to the main program). The declaration takes the form:

MODULE module-name;

(where module-name identifies the non-program file, and does not need to be the same as the name of the main program) or:

MODULE;

If a file does not contain the main program, there are some restrictions on the elements that it can contain. For information on the format of a non-program file, see Section 5.1, "Program Structure."

6.2 LABEL DECLARATIONS

If you want to transfer control to a particular section of a program, you must label that section with a "statement label." Labels are unsigned integers from 0 to 32767, and must be declared in a label declaration statement. The label declaration statement takes this form:

LABEL one or more numbers, separated by commas;

For example, if we want to use the labels 25 and 100 in a program, the declaration looks like:

LABEL 25, 100;

Labels appear in the program in front of the statement they designate, and end with a colon. For example:

```
25:     IF EOF THEN WRITELN('End of file.');
```

To reference a labeled statement, use the GOTO statement. (For information on GOTO, see Section 9.4, "GOTO.")

In addition to the standard labels we talked about above, AlphaPascal also recognizes another type of label which appears after the BEGIN and END keywords. The purpose of these labels is to enlist the compiler's help in determining whether or not you are properly nesting BEGIN-END blocks. If the same label appears after two BEGIN and END keywords, the compiler checks to make sure that the keywords do indeed mark the beginning and end of a block; if they do not, the compiler reports an error ("[STMBID] Wrong BEGIN-END identifier -- XXX expected," where XXX is the block label expected). This helps you to make sure that the structure of your program is correct. An example may help to clarify. Look at the following program diagram:

```
BEGIN : Label1
          .
          .
  BEGIN : Label2
          .
          .
  END : Label2
          .
          .
END : Label1
```

The example above shows a program in which the blocks are properly nested. By including the labels "Label1" and "Label2", we have asked the compiler to check the program structure and make sure that the BEGIN and END keywords are indeed nested properly. The program below will cause the compiler to report an error:

```
BEGIN : Block1
          .
          .
  BEGIN : Block2
          .
          .
END : Block1
          .
          .
  END : Block2.
```

since the END keyword for Block1 appears before the END keyword of Block2. The BEGIN-END label may take the form of any legal identifier, and must be separated from the keyword by a colon.

6.3 CONSTANT DEFINITIONS

Defining constants will be helpful whenever: you have a string or numeric literal that is used frequently within a program; a literal is important to understanding the logic of the program; or a literal may possibly be changed in future versions of the program. (For information on constants, see Section 8.2, "Constants.")

The constant definition takes the form:

```
CONST     identifier1 = number or string;
          identifier2 = number or string;
                      .
                      .
                      .
          identifierN = number or string;
```

For example, instead of repeating the expression "Radius * 3.1415927" throughout a program, you might want to define the constant Pi:

```
CONST   Pi = 3.1415927
```

Then, wherever your program used to say "Radius * 3.1415927", you can now say: "Radius * Pi". This keeps your program easy to read. Also, if at a future date you have to change a literal in your program, it is now a simple matter since you have only to change one constant definition statement instead of every occurrence of that literal in the program.

As an example of a string literal, consider the statement:

```
WRITELN('You have entered an invalid number-- try again');
```

If you use this string more than once, you might want to replace it with a constant:

```
CONST Error = 'You have entered an invalid number-- try again');
```

Now your statements can read:

```
WRITELN(Error);
```

6.4 TYPE DECLARATIONS

The most important feature of Pascal is its use and definition of the concept of "data types." A data type is a set of data (for example, whole numbers) that are alike in some way. For more information on data types, see Chapter 7, "Data Types." For now, let's just say that Pascal gives you some very powerful ways of representing different kinds of data types. Besides the standard types that Pascal recognizes (for example, the type INTEGER, that represents whole numbers), Pascal also allows you to define your own data types. You must declare a user-defined data type at the front

of the main program or procedure in which you are going to access that data type. The type declaration takes this form:

```
TYPE    identifier1 = type1;
        identifier2 = type2;
                .
                .
                .
        identifierN = typeN;
```

For example, suppose you want to define a new data type that is a simple scalar type whose elements are: MON, TUES, and WEDS. You can do so by simply enumerating the elements of that type:

```
TYPE  Days = (MON,TUES,WEDS);
```

On the other hand, suppose you want to declare a more complicated data type, such as a type of array:

```
TYPE  NewArray = ARRAY [1..10] OF INTEGER;
```

The declaration above declares an array named NewArray which contains 10 elements (which are to be indexed by the numbers 1 through 10). The elements are of type INTEGER.

6.5 VARIABLE DECLARATIONS

Pascal requires that all variables be "declared." This means that you assign a name to a variable and permanently associate a data type to that variable. Since you tell Pascal the data type of each variable, Pascal knows what operations can be performed on that variable, and which functions and procedures can be used on it.

Be aware that Pascal does not assume an initial value (e.g., zero) for a declared variable; you must explicitly assign a value to a variable. If you try to assign a value that is not consistent with the data type associated with that variable, the Pascal compiler generates an error message.

The variable declaration statement takes the form:

```
VAR     identifier...,identifier : data-type;
        identifier...,identifier : data-type;
                        .
                        .
                        .
        identifier...,identifier : data-type;
```

For example:

```
VAR     TestScores, Variance, Mean : REAL;
        StudentID, ClassName,
              StudentName, Teacher : STRING;
                           Passed : BOOLEAN;
```

The variable name may be any legal identifier. The data types you can assign to a variable are discussed in Chapter 7, "Data Types."

6.6 FUNCTION AND PROCEDURE DECLARATIONS

You may often need to perform the same sort of actions on a body of data throughout your program. Rather than forcing you to tediously duplicate one piece of code every place it is needed, Pascal gives you two ways to generate "subprograms" which may be called upon wherever needed in a program. These subprograms are called "functions" and "procedures." Such subprograms also help you to maintain your programs since, if a change must be made, it only needs to be made once.

Although you may invoke these functions and procedures any place in the statement part of your program (or within the declarations of other functions and procedures), you must first define the functions and procedures within the declaration part of your program before you invoke them. (A special case exists for referencing functions and procedures within other functions and procedures before they have been defined; see Section 6.6.3, "Forward Declarations.")

Functions and procedures can be thought of as programs within a program. They can declare variables, define and invoke procedures and functions of their own (known as "local" procedures and functions), and input and output data.

6.6.1 Functions

A function is a subprogram that performs some computation and returns a value. (For example, the standard function ABS takes a number and returns the absolute value of it.) Pascal allows you to define your own functions by including function declarations at the front of the program or procedure that will call that function. Function declarations must appear after any variable declarations.

The function declaration takes this form:

```
FUNCTION function-name (formal parameters) : data-type of result;
        function-block;
```

where the formal parameters are identifiers that describe the variables (and their data types) which will be used within the function. These variables do not have to appear in a variable declaration statement, since they are being declared within the function heading.

Following the formal parameters is the data type of the result of the function. For example:

```
FUNCTION SufficientFunds(Request : REAL) : BOOLEAN;
   BEGIN
      SufficientFunds := Request <= AmountAvailable
   END;
```

The heading above might identify a function that returns TRUE if a checking account has enough funds to cash a specified check. The function block starts with the BEGIN keyword and finishes with the END keyword. The statements in between perform the action on the input data when the function is executed. The function block takes this form:

```
Label declarations
Constant declarations
Type declarations
Variable declarations
Procedure/function declarations
BEGIN-END block
;
```

As you can see, the block of the function follows much the same form as the program block itself, except that a function definition ends with a semicolon, rather than a period. At some point within the BEGIN-END block, a value must be assigned to the function name itself. This is the way that the result of the function is returned to the program or procedure that invoked it.

To invoke a function, include the name of the function within the program block along with the names of the variables that are going to supply that function with data. For example, to invoke the function SufficientFunds, you might include a statement line like this:

```
IF SufficientFunds(100.50)
    THEN WRITELN('Good check')
    ELSE WRITELN('Sorry, overdrawn');
```

The statement above prints 'Good Check' if SufficientFunds returns TRUE, and 'Sorry, overdrawn' if it returns FALSE. You may supply variables, expressions, or constants as the arguments of the function. Note that the names of the variables you pass to the function do not have to have the same names as those variables listed in the function heading. The first variable (or constant) mentioned in the function invocation is substituted into the function for the first variable mentioned in the function heading, the second variable (or constant) in the invocation replaces the second variable in the function heading, and so on. (Of course, the data types of the variables must be consistent. For example, if you supply the variable Check to the function SufficientFunds, it must contain a number of type REAL.)

Remember that a function invocation is always part of an expression. For example, given the function MaxNum, these are valid function invocations:

```
WRITELN('The largest number is: ',MaxNum(Number1,Number2));
```

or:

```
IF MaxNum(Value1,Value2) < 0 THEN WRITELN('Numbers are negative.');
```

Let's look at an example of a function and function invocation. Suppose your program frequently needs to check the range of input numbers. A simple function to make sure that a number is between 1 and 100 might look something like this:

```
PROGRAM Validate; { Validate a numeric entry; make sure
                    that it is between 1 and 100. }

VAR   Target : REAL;

FUNCTION ErrCheck(Local : REAL) : BOOLEAN;
 { Function does error checking on entry. If 100 < number < 1,
   ErrCheck reports error by returning a TRUE. }
  BEGIN { Begin function ErrCheck }
    ErrCheck := Local < 1 OR Local > 100
  END { End function ErrCheck };

BEGIN { Main Program }
        WRITE('Enter a number between 1 and 100: ');
        READLN(Target);
        IF ERRCHECK(Target)
           THEN WRITELN('Invalid entry: try again.')
           ELSE WRITELN('Very good. Correct entry.')
END { Main Program }.
```

Note that until the program begins executing the main program, where the function is actually invoked, the function is not executed, even though the function definition appears at the front of the program.

6.6.2 Procedures

The major purpose of a function is to compute and return a value. The main purpose of a procedure is to perform a set of operations. For example, let's say that you are designing a program that plays a card game. At various times throughout the program you may need to simulate the shuffling of a deck of cards. Rather than include this same piece of code throughout your program (which would make the program hard to read and maintain), you may designate this piece of code as a procedure. The procedure declaration names the procedure, tells what kinds of variables it will use, and gives the statements that make up the procedure. It takes this form:

```
PROCEDURE procedure-name (formal parameters);
        procedure-block;
```

The formal parameters list the variables (and their types) with which the procedure will work. For example:

```
PROCEDURE PrintReport (Title : STRING; PageSize : INTEGER);
```

The procedure block takes this form:

```
Label declarations
Constant declarations
Type declarations
Variable declarations
Procedure/function declarations
BEGIN-END block
;
```

To invoke the procedure, include the name of the procedure within your program. Unlike a function invocation, a procedure invocation is a program statement, not an expression. For example, say that you have a procedure named Shuffle that simulates the shuffle of a deck of cards:

```
BEGIN
  IF Dealer = New OR Deck = Empty
    THEN Shuffle
END;
```

Although a procedure may take a form very much like that of a function, it does not necessarily return a value. Notice that it also does not have to accept any arguments. (For information on using procedures to return several results, see Section 6.6.4.2, "Reference Parameters.")

6.6.3 Forward Declarations

What happens when a procedure or function declaration invokes a procedure or function whose declaration has not yet appeared in the program? There are times when for aesthetic or practical reasons (or because the two routines call each other) you must invoke a procedure or function before its definition appears in the declaration part of the program. Pascal provides a way to do this.

The forward declaration tells the Pascal compiler, "We'll define this later; don't worry that you haven't seen its declaration yet." The forward declaration takes the same form as the heading of a procedure or function declaration, except that the word FORWARD replaces the procedure or function block. In effect, we separate the heading from the block. For example, take a look at the procedure DrawLine:

```
PROCEDURE DrawLine (Character : CHAR; LineSize, Angle : REAL);
  FORWARD;
```

Now a function or procedure declaration may appear that invokes the procedure or function. Later within the declaration part of the program, the actual procedure or definition block appears, preceded by the name of the function. For example:

```
PROGRAM TaxReturn;  { This program computes tax returns.  First it
                      asks if the user wants instructions (short or long). }

VAR     Short : BOOLEAN;
        Query : CHAR;

PROCEDURE Display(Short : BOOLEAN);
  BEGIN { Display }
     { This is the procedure that actually displays the
       instructions.  It prints a long or a short
       file, depending on the value of Short. }
                          .
                          .
  END { Display };

PROCEDURE PrintInstructions (Short : BOOLEAN);
FORWARD;        { The forward reference! }

FUNCTION AskAnswer (Query : CHAR) : BOOLEAN;
  BEGIN { AskAnswer }
    AskAnswer := FALSE;
    IF Query = 'Y' OR Query ='y' THEN AskAnswer := TRUE
      ELSE IF Query = '?' THEN PrintInstructions(Short);
  END { AskAnswer };

PROCEDURE PrintInstructions;
  BEGIN { PrintInstructions }
     Short := FALSE  { Initialized to long instructions. };
     WRITE('Do you want short Instructions? Y or N:');
     READLN(Query);
     IF AskAnswer(Query) THEN Short := TRUE;
     Display(Short)
  END { PrintInstructions };

BEGIN { Main Program }
   WRITELN('We''re going to compute your tax return.'); WRITELN;
   WRITELN('At any time in this program, you may
      review the instructions');
   WRITELN('by answering any Y or N question with a ''?''.');

   WRITE('Do you want instructions? (Y or N): '); READLN(Query);
   IF AskAnswer(Query) THEN PrintInstructions(Short);
   WRITE('Do you want to average? (Y or N): '); READLN(Query);
   IF AskAnswer(Query) THEN WRITELN('OK, We''ll average.');
   { Now, compute taxes... }
                          .
                          .
END { Main Program }.
```

Note that when the procedure block PrintInstructions appeared after the Function AskAnswer, we did not include the formal parameters for that procedure, since the procedure heading appeared at the time of the forward reference.

6.6.4 Formal Parameters

We would like to include a word here on formal parameters. Parameters are variables used within a function or procedure. Pascal greatly extends the usefulness of your routines by allowing your program to supply those values at the time that you invoke your function or procedure. This means that you can use your routines in a wide variety of situations, on a wide range of data. Parameters give your functions and procedures a way to communicate with the program that calls them.

The variables that are specified at the time you define your function or procedure are called the "formal parameters." The values you supply with the actual invocation of your routine are called the "actual parameters." For example, given the function heading:

```
FUNCTION Salary(Takehome, Gross : REAL) : REAL;
```

the formal parameters are the variables Takehome and Gross. When we invoke that function we might do so using constants:

```
Raise:=Salary(183,250);
```

or, we might use variables which contain those values

```
Raise:=Salary(Net,Total);
```

Note that the variable identifiers we use as formal parameters do not have to be the same as the identifiers for the actual parameters. You can think of the formal parameters as "placeholders" for the actual data which will be used. The actual parameters are "plugged into" the formal parameters in the same order as they appear in the routine invocation. (For instance, in the example above, Net takes the place of Takehome, and Total takes the place of Gross.) The total number of actual parameters must match the number of formal parameters.

6.6.4.1 Value Parameters - The formal parameters we have seen in our examples above were all used to pass information into the function or procedure. When we left the function or procedure, the value of the variable we passed into the routine was not actually changed, even though it might have been modified within the routine. In effect, the function or procedure made a copy of the variable and used the copy for its calculations. Then when we left the routine, the original value of the variable was unchanged.

This type of variable is called a "value parameter." Value parameters may be variables or expressions.

6.6.4.2 Reference Parameters - It sometimes happens that you would like a procedure or function to actually modify a variable. (Otherwise, the only values you could return would be the single value returned by a function.) To tell a function or procedure not to use a copy of a variable, but to use the variable itself, include the VAR keyword in front of the parameter. For example:

```
FUNCTION Justify(VAR InputString:STRING;PageWidth:REAL):REAL;
```

which might modify the string InputString by inserting blanks so that it equaled PageWidth in length, and returns the number of blanks inserted. A parameter like InputString is called a "reference parameter."

Another way of looking at value parameters and reference parameters is that in the case of value parameters we are really dealing with two different sets of variables: those outside the routine and those inside. In the case of reference parameters, we are dealing with only one set of variables. Reference parameters must be variables.

6.7 EXTERNAL DECLARATIONS

AlphaPascal provides an external library of procedures and functions. This collection of useful routines is available for use by your program. You may also write your own external libraries. To tell AlphaPascal that you are going to use a function or procedure that is in a standard library other than STDLIB, you must precede the declaration of that function or procedure with the keyword EXTERNAL. For example:

```
EXTERNAL FUNCTION Graph (X,Y : REAL) : REAL;
```

or:

```
EXTERNAL PROCEDURE PrintLine (Line : STRING);
```

You do not include the procedure block or function block, since the actual definition of the routine is in the external library.

Besides identifying procedures and functions within your program that are defined in an external library, you will use the external declaration to designate elements that appear in files that are not a main program file. For example, suppose you have a main program file and three other files which will be linked together to form one program. (See Section 5.1, "Program Structure," for information on main program and non-program files.) Within one file you may well want to use a procedure, function, or variable that was declared and defined in another file. If you are going to link a number of files together into one program, each file must contain an

external declaration for every element it needs to reference, if that element was declared and defined in another file.

For example, if the variable CustomerID was declared in file File3, and you need to reference that variable in File2, File2 contains the external declaration:

```
EXTERNAL VAR    CustomerID : STRING;
```

There are some things you should keep in mind when making external declarations:

1. You may not externally declare labels, constants, or types. If you need to have common definitions of these items, use include files. For information on include files, see Section 4.3.2.2, "The Include Option ($I)."

2. If you are going to use data files in your program, the declarations for those data files must be in the main program file. (That is, data files may not be externally declared in your main program file.)

3. You must be very sure that the types given in your external declarations exactly match the types given in the original main declarations. For example, if one file has the declaration:

   ```
   VAR NetWork : CHAR;
   ```

   the external declaration in another file for that variable must specify type CHAR:

   ```
   EXTERNAL VAR NetWork : CHAR;
   ```

CHAPTER 7

DATA TYPES

We've already mentioned that a variable is a symbol that can represent more than one data value. We've also said that you must "declare the type" of each variable used in a program. This chapter discusses the idea of "data type," and the various data types available in Pascal.

A data type describes the kinds of values that a variable can assume. For example, if the variable CustomerID can assume only numeric, integer values, we say that its data type is "integer." Some languages allow you to let one variable assume a variety of types. (For example a variable could have the integer value 34 at one point, and the real value 34.56 at another point.) Pascal, on the other hand, allows each variable to assume only one kind of data type.

Pascal requires that you declare the type of data that a variable can assume. This results in several advantages: 1) you can always deduce the type of values a variable can assume by reading the program; you do not have to run the program to figure it out; 2) certain operations may only be done on specific data types; having to declare your variables aids the compiler in making sure that you are not performing an illegal operation on a variable; 3) the compiler is able to make sure that you are not improperly mixing variables of different data types. (For example, you may not multiply a real number by an integer and get an integer result.) Once a variable has been assigned a data type, we have automatically defined the operations that can be applied to that variable, the type of values it can assume, and the standard procedures and functions that can be used on it.

Several data types have been pre-defined for you by AlphaPascal; these are called "standard data types." The AlphaPascal standard data types are: INTEGER, REAL, BOOLEAN, CHAR, STRING, and TEXT.

Data types are grouped into two categories: simple and structured. A simple data type is a "scalar" type. A scalar data type is one that contains a set of elements, and those elements are ordered. For example, the INTEGER data type contains the set of whole numbers. These elements are ordered; for instance, -2 is less than -1 which is less than 0 which is less than 1 which is less than 2, and so on.

Structured data types are more sophisticated than the simple, scalar data types. If you were to create your own structured types, they would be made up of simple data types. Pascal supplies a set of keywords (SET, ARRAY, RECORD, and FILE) that you can use to build structured types.

7.1 SIMPLE DATA TYPES

Simple data types can either be the pre-declared simple data types (INTEGER, REAL, BOOLEAN, and CHAR), or they may be types defined by you. If defined by you, a simple data type is either a scalar type or a subrange of another, already defined scalar data type.

7.1.1 INTEGER

Integers are whole numbers (that is, numbers with no fractional part). AlphaPascal allows you to use integers in the range of -32767 through 32767. They are stored by the computer as one-word, signed 2's complement binary numbers. These are integers:

```
32000
0
1
-450
MAXINT
+56
```

(Remember that the pre-declared constant MAXINT is the largest integer that AlphaPascal can represent, 32767.)

The standard identifier INTEGER designates the integer data type. For example:

```
VAR   Ellipse, Counter, Control : INTEGER;
```

The operators that have been defined for integers are: addition (+); subtraction or sign inversion (-); multiplication (*); integer division-- that is, divide and truncate-- (DIV); modulus (MOD); the set membership operator IN; and, the relational operators. Using other operators (for example, the real division operator, /) on integers causes the compiler to generate an error message.

There are many functions that accept INTEGER arguments. (See Chapter 12, "Mathematical Functions," for a list of the trigonometric, hyperbolic trigonometric, and mathematical functions.)

Two other functions often used on INTEGER data are the PRED and SUCC functions. PRED returns the predecessor element of the data type; SUCC returns the successor element of the data type. For example, given three variables, ONE, TWO, THREE of type INTEGER, and ONE = 1, TWO = 2, and THREE = 3: PRED(TWO) returns 1; SUCC(TWO) returns 3. (See Sections 11.1.6 and

11.1.8 for information on PRED and SUCC.)

7.1.2 REAL

Real numbers are decimal numbers that may contain a fractional part. As noted in Section 5.6, "Notation," we can represent real numbers either in decimal notation or in scientific notation. These are real numbers:

```
9.3
-56.7812
7.03E+5
+45.0
1.03E-3
```

The computer stores real numbers as three-word floating point numbers significant to 11 digits (12 for real numbers in which the fractional part is zero or less than 1E12), with an exponent range of roughly 1E-37 to 1E37.

The standard identifier REAL designates the real number data type. For example:

```
VAR   Mean, Median, Variance : REAL;
```

The operators defined for real numbers are: addition (+); subtraction and sign inversion (-); multiplication (*); real division (/); and, the relational operators. Many functions accept REAL numbers as arguments. Note that you may not use the PRED and SUCC functions or the set membership operator IN on REAL data.

7.1.3 BOOLEAN

The Boolean data type contains two elements: TRUE and FALSE. These elements are ordered so that FALSE < TRUE. (And, SUCC(FALSE) returns TRUE.) FALSE and TRUE are pre-declared constants. A Boolean variable represents a logical true or false value. For example:

```
IF Month = April THEN Spring := TRUE
```

In the statement above, Spring is a Boolean variable that can assume the values TRUE or FALSE.

To designate a Boolean data type, use the standard identifier BOOLEAN. For example:

```
VAR   Query, Female, Employee : BOOLEAN;
```

The operators defined for Boolean data are: AND, OR, and NOT. These are called Boolean operators, and produce a Boolean result. For example:

```
X AND Y
```

gives a result of TRUE if both X and Y are TRUE, or FALSE if either X or Y (or both) are FALSE.

When we use the relational operators on INTEGER, REAL, CHAR, or STRING data types, the result is always of type BOOLEAN.

You may use the PRED and SUCC functions on data of type BOOLEAN, and you may use the set membership operator, IN. You may also use the ORD function:

```
ORD(FALSE) = 0
ORD(TRUE) = 1
```

7.1.4 CHAR

The computer recognizes a specific set of characters that it can represent. The elements of this set are ordered; for example, A < B < C... In the case of the Alpha Micro computer, this ordering is called the "ASCII collating sequence," and the set of characters is called the "ASCII character set." (For a list of the ASCII characters, see Appendix B, "The ASCII Character Set.")

A CHAR variable contains one ASCII character. To indicate an element of CHAR data type, enclose it in single quotes. For example:

```
VAR   MenuChoice : CHAR;

MenuChoice := 'A';
```

The relational operators have been defined for use on CHAR data. Remember that A < B because of their position in the ASCII collating sequence. You may also use the set membership operator, IN on data of type CHAR.

To designate data as type CHAR, use the CHAR standard identifier.

```
VAR Initial : CHAR;
```

or:

```
TYPE  Character = CHAR;
VAR   Item : Character;
```

Because CHAR is a non-REAL scalar type, you can use the SUCC and PRED functions to identify predecessor and successor elements of the type. For example:

```
PRED('B')
```

returns an 'A'. You can also use the ORD function to determine the position of the character in the ASCII character set. (For more information on PRED, SUCC, and ORD, see Chapter 11, "Miscellaneous Functions and Procedures.")

NOTE: Remember that CHAR data is only one ASCII character. Another standard data type exists, STRING, which represents a collection of CHAR data. For example: 'A' is CHAR data, but 'ABCD' is STRING data. For information on STRING, see Section 7.2.3, "STRING."

7.1.5 User-Defined Scalar

Pascal allows you to define your own scalar types. To do so, use the type declaration statement. You will supply the name of the data type, and the elements of which it is composed. For example:

```
TYPE Spectrum = (Violet,Blue,Green,Yellow,Orange,Red);
```

Just like any other scalar type, your data type consists of ordered elements. This ordering is reflected by the order in which you list the elements in the type declaration statement. For example, given the statement above, Violet < Blue < Green, and so on. You can then declare and use a variable of the data type you have defined. For example:

```
VAR Colors : Spectrum

IF Colors = Red THEN WarmColor := TRUE;
```

The relational operators have been defined for user-defined scalar types, and return a Boolean result. Internally, the computer stores each of these elements as an integer value. (For example, in the example above Violet is 0, Blue is 1, and so on.)

You may not use scalar types in I/O operations. For example, this statement is illegal:

```
WRITE(Yellow)
```

if Yellow is an element of a user-defined scalar type. However, you could say something like:

```
IF Colors = Yellow THEN WRITE('Yellow');
```

Note that Colors is a variable, but Yellow is a constant of the scalar type Spectrum (just as the number 2 is a constant of the scalar type INTEGER). You may only use relational operators and the set membership operator, IN, on an element of a user-defined scalar type.

NOTE: Rather than using a type declaration followed by a variable declaration, you may combine both statements into one variable declaration when defining your own data types. For example:

```
VAR WaveLengths,Colors : (Violet,Blue,Green,Yellow,Orange,Red);
```

However, if you are going to have more than one variable declaration that declares variables of that type, you must have a separate type declaration statement instead.

You may use the ORD, PRED and SUCC functions on user-defined scalar types. For example, given our example above:

```
ORD(Violet) = 0
ORD(Blue) = 1
SUCC(Violet) = Blue
PRED(Orange) = Yellow
```

7.1.6 User-Defined Subrange

Pascal allows you to define a subrange of a previously defined data type. For example, given the data type Spectrum above, suppose you want a variable to only access the first three colors elements of that type, Violet, Blue, and Green. You could define a subrange scalar type:

```
TYPE ColdColors = Violet .. Green;
```

You may define a subrange of any user-defined or standard scalar type except type REAL. Use the type declaration statement in this format:

```
TYPE Type-name = lowerlimit .. upperlimit;
```

The symbols ".." tell Pascal that you are establishing a subrange. Upperlimit and Lowerlimit are the beginning and ending elements of the subrange. For example:

```
TYPE Decimal = '0' .. '9';
```

tells Pascal that we want to define a type named Decimal that can assume values in the range of '0' through '9' of the standard data type CHAR. We can then declare a variable of that type:

```
VAR Number : Decimal;
```

NOTE: You may also directly declare a variable to of a subrange without using a type declaration statement. For example:

```
VAR Number : '0' .. '9';
```

7.2 STRUCTURED DATA TYPES

Structured data types are built up of simple scalar data types. Several keywords can be used to define structured data types: ARRAY, RECORD, SET, and FILE.

You may define your own structured data types in much the same way that you were able to define simple scalar types. (See Section 7.1.5, "User-defined Scalar.") Two structured types have been pre-declared for you: STRING and TEXT.

7.2.1 Packed Data Types

Before we discuss the various structured data types available to you, we'd like to digress for a moment and talk about how the computer represents data types in memory.

Structured data types sometimes require quite a bit of room in memory. For example, consider how many memory locations must be allocated for a structure such as:

```
ARRAY [0..10,0..10,0..10,0..10] OF CHAR;
```

where more than 10,000 elements must be handled. (NOTE: We discuss the ARRAY data type in Section 7.2.2, "ARRAY.") It is often the case that only one element of such a structure is stored in one memory location, even though there physically may be room for more. To help minimize memory use, Pascal allows you to create "packed" data structures, in which the data in the structure are packed together in a minimum amount of space. To create a packed data type, include the keyword PACKED in your type declaration statement:

```
TYPE Type-name = PACKED data type
```

For example:

```
TYPE CustomerID = PACKED ARRAY [1..50] OF CHAR;
```

You may also pack records by preceding the keyword RECORD with the word PACKED. Only the array or record immediately following the PACKED keyword is affected, and any nested arrays or records must be explicitly packed. As one example of the efficiency you can sometimes gain in packing data, consider the following data structure of type RECORD:

```
TYPE     Date =
             RECORD
               Month : (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sept,Oct,Nov,Dec);
               Day   : 1..31;
               Year  : 0..99
             END;
```

Unpacked, the data above takes up three words of memory; packed, it takes up only one word.

NOTE: Some types of data cannot be packed (e.g., real numbers), and the keyword PACKED in the type declaration for such data types has no effect.

Your program does not need to handle a packed data type differently than any other data type. (NOTE: Standard Pascal requires that you use the UNPACK and PACK standard procedures to convert between packed format and a format that your program can read and write. AlphaPascal performs this conversion for you automatically. In fact, AlphaPascal does not support the PACK and UNPACK procedures.)

Although you do save memory space by packing a data type, be aware of the fact that your program will run slower when it handles such a data structure, because of the time required to unpack and repack data.

7.2.2 ARRAY

An array has a fixed number of components which may be accessed in any order by referencing the location of the element within the array. To reference an element of the array, you give the name of the array, and the array index (sometimes called a subscript) which selects the location within the array whose contents you want to access. The subscript appears after the array name in square brackets:

```
Array-name[ Index1, Index2, ... IndexN ]
```

where each index is a simple type. For example, suppose the array PartNos contains thirty part numbers, and you want to see what the twentieth one is. You would access the twentieth location in the array by saying:

```
WRITELN(PartNos[20]);
```

or perhaps:

```
WRITELN(PartNos[2 + Offset];
```

All elements of an array must be of the same data type. Your declaration of the array must include the data type of the elements of the array, and the data type of the subscripts by which you will access elements of that array. (Declaring the type of the subscript tells Pascal how many elements the array will contain.) For example:

```
TYPE MonthTotals = ARRAY[1..20] OF REAL;
```

The statement above tells Pascal that you are defining an array type named MonthTotals whose elements will be real numbers, and that the locations in that array will be accessed by refering to the numbers 1 through 20 (e.g., MonthTotals[1], MonthTotals[2], ... MonthTotals[20]).

The subscript data type can be any scalar type except REAL. Although this field will often be of type INTEGER, it doesn't have to be. For example:

```
TYPE ComplaintNum = ARRAY [BobsOffice .. PaulsOffice] OF INTEGER;
```

where BobsOffice..PaulsOffice is a subrange of a user-defined scalar type, such as (RobinsOffice, BobsOffice, PaulsOffice, BillsOffice).

After you have declared an array type, you may now declare a variable of that type. For example:

```
VAR Problems : ComplaintNum;
```

Pascal also allows a shorthand form that permits you to combine the type and variable declarations:

```
VAR Problems : ARRAY [BobsOffice .. PaulsOffice] OF INTEGER;
```

One of the features that help make arrays so useful is the fact that subscripts may be expressions. This allows you to access elements of the array using variables for the subscripts. For example:

```
PROGRAM SquareIt;

VAR     Square : ARRAY[1..10] OF INTEGER;
       Counter : INTEGER;

BEGIN { SquareIt }
  Counter := 1;
  WRITELN('Squares of the integers 1 to 10 are: ');
  FOR Counter := 1 TO 10 DO
    BEGIN
      Square[Counter] := Counter*Counter;
      WRITELN(Square[Counter])
    END;
END { SquareIt }.
```

The small program above creates array Square of ten elements. The FOR-DO loop increments the variable Counter from 1 to 10, accesses the array location indexed by Counter, and writes the square of Counter into that location of the array. (For example, location Square[5] contains the number 5*5, or 25.) You can use a similar type of loop to retrieve data from an array. NOTE: Sometimes you can fill an array without using loops. For example:

```
InvoiceNum['A'] := InvoiceNum['B'];
```

accomplishes the same thing as:

```
FOR I := 1 TO 5 DO InvoiceNum['A',I] := InvoiceNum['B',I];
```

7.2.2.1 Multi-dimensional Arrays - Until now our discussion has been of "one-dimensional" arrays; that is, arrays with just one index. Pascal also allows you to construct arrays with an unlimited number of dimensions. (You might consider a multi-dimensional array as an "array of an array.") To declare such a structure, include additional subscripts in the declaration. Suppose you want to keep track of a five-element array, each element of which is in turn a five-element array:

```
TYPE InvoiceNums ARRAY['A'..'E'] OF ARRAY[1..5] OF INTEGER;
```

Pascal also allows a shorthand form:

```
TYPE InvoiceNums ARRAY['A'..'E',1..5] OF INTEGER;
```

The statements above create a two-dimensional array of 25 elements. Each element is referenced by a pair of subscripts. If we wanted to make a pictorial representation of our array InvoiceNums, it might look something like this, with the Xs representing integer numbers contained in the array:

	1	2	3	4	5
A	X	X	X	X	X
B	X	X	X	X	X
C	X	X	X	[?]	X
D	X	X	X	X	X
E	X	X	X	X	X

If we wanted to access any number in the array, we would have to specify the subscripts that designate the proper location. (In the example above, 'A'..'E' designate array "rows"; 1..5 designate array "columns.") The subscripts for a two-dimensional array must identify the element's row and column. For example, to identify the element marked with a question mark in the table above, we would ask for Row C, Column 4:

```
InvoiceNums['C',4]
```

The number of dimensions an array may contain is limited only by the room in memory.

7.2.3 STRING

We have already mentioned the data type CHAR. A variable of type CHAR contains a single ASCII character. However, we often need to refer to collections of characters (such as words, names, or addresses) rather than just single characters.

The standard data type STRING allows you to declare variables that contain a group (or "string") of ASCII characters. For example:

```
VAR   AccountID : STRING;
```

The default maximum string length is 80 characters, but you can set the string length maximum to from 1 to 255 characters. To set maximum string length, follow the identifier STRING with an integer constant in square brackets. For example:

```
TYPE OrderID = STRING[25];
```

The STRING data type is approximately equivalent to:

```
TYPE STRING[N] = PACKED RECORD
                   LEN : 0..255;
                   TXT : ARRAY [1..N] OF CHAR;
                 END;
```

If N above is omitted, STRING defaults to size 80. (NOTE: The structure given above for STRING is approximate, and is only given for illustrative purposes; you cannot access the length of string X by referring to X.LEN.) The computer stores strings with one character per byte, and one byte at the front of the string which tells Pascal how long the string is.

7.2.4 TEXT

The standard data type TEXT is equivalent to the type FILE OF CHAR. For example, suppose you want to declare and open a text file, you could say:

```
PROGRAM ReadListing;

TYPE  ListFile = TEXT;

VAR   ProgramList : ListFile;

BEGIN { ReadListing }
   OPEN(ProgramList,'ACCNT1.DAT',OUTPUT);
                .
         { read data from file }
                .
END { ReadListing }.
```

NOTE: In the example above, it would also have been valid just to say: VAR ProgramList : TEXT. (Note to users of previous versions of AlphaPascal--the file type INTERACTIVE is no longer needed or supported. Replace any occurrences of the identifier INTERACTIVE with TEXT, or at the front of your program re-define INTERACTIVE (e.g., TYPE INTERACTIVE = TEXT).) (For information on type FILE, see Section 7.2.6, below.)

7.2.5 SET

Sets give you a very efficient way of handling certain kinds of information. Although, they are not exactly analogous, you might think of sets as a kind of packed Boolean array. The use of sets allows complex logical expressions to be written concisely, and also gives a more flexible way of performing logical tests. For example, instead of the cumbersome statement:

```
IF (Character = 'A') OR (Character = 'B') OR
        (Character = 'C') OR (Character = 'D')
          OR (Character = 'E')
    THEN Flag := TRUE;
```

using sets, you can simply say:

```
IF Character IN ['A'..'E'] THEN Flag := TRUE;
```

To define a set type, use the type declaration statement. Every element of the set must be of the same type, and that type may not be structured. You must specify the name of the set data type, and the base type of that set:

```
TYPE Identifier = SET OF base-type;
```

For example:

```
TYPE Player = SET OF 1..5;
```

Once you have defined the set, you can now declare a variable of that type:

```
VAR Piece : Player;
```

which can assume one or more of the values of that set. Pascal also allows a shorthand declaration:

```
VAR Piece : SET OF 1..5;
```

The symbol [ ] is the set constructor operator. It takes a list of expressions of the form:

```
[expression]
```

or:

```
[expression .. expression]
```

For example, given that Y is of type SET, the following is a valid assignment statement:

```
Y := [X, X+5 .. X+7];
```

It assigns the element X and the elements X+5 through X+7 to the set Y. You may mix sets of the same base type. For example:

```
VAR
     X : SET OF 'A'..'X';
     Y : SET OF 'L'..'Z';

BEGIN
     Y := Y + X;
END.
```

You may use modifying assignment operators on sets. (So, for example, you could rewrite the statement above to: Y += X;.)

The operations that you can perform on a set are those defined by set theory: set union (+); set difference (-); set intersection (*); set equality (=); set inequality (<>); set inclusion (<= and >=); and, set membership (IN). The empty set, "[]", is a valid set.

If we define a type Newset that is a set of integers:

```
TYPE NewSet = SET OF 1 .. 10;

VAR      Set1 : NewSet;
         Set2 : Newset;
       Result : Newset;
```

and then assign values to the sets Set1 and Set2:

```
Set1 := [1..5];
Set2 := [5,6,7,8,9];
```

We can use the sets Set1, Set2, and Result to talk about the operations you can perform on sets:

\+ Set Union. An element is contained in the union of SET1 and SET2 if and only if it is an element of SET1 or SET2 or both. For example:

```
Result := Set1 + Set2    { Result is the set [1..9] }
```

\- Set Difference. An element is contained in the difference of two sets if and only if it is an element of SET1 but not an element of SET2. For example:

```
Result := Set1 - Set2    { Result is the set [1..4] }
```

* Set Intersection. An element is contained in the intersection of two sets if and only if it is an element of both SET1 and SET2. For example:

```
Result := Set1 * Set2          { Result is the set [5] }
```

= Set Equality. Set1 = Set2 is TRUE if and only if every member of Set1 is also a member of Set2, and every member of Set2 is also a member of Set1.

```
Result := Set1 = Set2          { Result is FALSE }
```

<> Set Inequality. Set1 <> Set2 is TRUE if and only if Set1 = Set2 is FALSE.

```
Result := Set1 <> Set2         { Result is TRUE }
```

<= Set Inclusion. The relation Set1 <= Set2 is TRUE if and only if every member of Set1 is also a member of Set2. In other words, Set1 <= Set2 is TRUE if Set1 is included in Set2.

```
Result := Set1 <= Set2         { Result is FALSE}
```

[6,9] <= Set2 is TRUE.

>= Set Inclusion. The relation Set 1 >= Set2 is TRUE if and only if every member of Set2 is also a member of Set1. In other words, Set1 >= Set2 is TRUE if Set2 is included in Set1. If X <= Y is TRUE, then Y >= X is TRUE.

IN Set Membership. If X is of the type declared as the base-type of Set1, then X IN Set1 is TRUE if and only if X is contained in Set1. For example:

```
Result := 5 IN Set1            { Result is TRUE }

Result := 26 IN Set1           { Result is FALSE }
```

The IN operator takes as a left argument a simple data type variable or constant (e.g., CHAR or INTEGER); the right argument must be a set of that data type (e.g., set of CHAR or set of INTEGER).

Below is a small sample program that uses sets:

```
PROGRAM;

VAR       Y1,Y2,Y3,N1,N2,N3 : CHAR;
                      Query : CHAR;
                   Yes,No : SET OF CHAR;
BEGIN
  Yes := ['Y']; No := ['N'];
  WRITELN('The only valid response to a Yes/No question is Y or N.');
  WRITELN('We''ll let you add your own answers.'); WRITELN;
  WRITELN('Enter three one-character symbols that can stand for YES ');
  WRITE('(separate them with a space, not a comma): ');
  READLN(Y1,Y2,Y3);
  Yes := [Y1,Y2,Y3] + Yes;  { Add user-defined symbols to Yes }
  WRITE('Now, enter three symbols for NO: ');
  READLN(N1,N2,N3);
  No := [N1,N2,N3] + No; { Add user-defined symbols to No }
  WRITELN;
  WRITE('Let''s test this out.  Enter a Yes or No answer: ');
  READLN(Query);
  WRITELN;

  IF Query IN Yes THEN WRITELN('Yes!')
     ELSE IF Query IN No THEN WRITELN('No!')
       ELSE WRITELN('I didn''t understand you.')
END.
```

7.2.6 FILE

A file is a structured data type that contains a sequence of elements of the same type. Since you can only access one element at a time, files might seem much like an array. The important difference is that file are associated with AMOS disk files, and so can store data permanently beween program runs. Files are the means of communicating with devices such as terminals and printers.

In addition, unlike other structured types, the size of a file does not have to be declared, and may be of any size supported by the AMOS file structure. Files typically hold data of type CHAR or they contain records (see Section 7.2.7, "RECORDS").

Use the type declaration to declare the data type:

```
TYPE  identifier = FILE OF base-type;
```

where identifier is the name you want to assign to that type of file, and base-type is the data type of the data in the file.

To use this type of file, you will have to define a variable of that type:

```
VAR   file-identifier : identifier;
```

The file-identifier acts as a communication channel. Using commands such as OPEN (see Section 10.2.12, "OPEN"), you can associate the file-identifier with an actual AMOS file, and transfer data between your program and the disk file.

Rather than using a type declaration followed by a variable declaration, AlphaPascal also permits you to use a shorthand method of combining type and variable declaration statements:

```
VAR   NewData : FILE OF INTEGER;
```

Remember that you must use one of the functions or procedures discussed in Chapter 10, "I/O Functions and Procedures," to tell AlphaPascal which AMOS file you want to associate with the file variable that you have declared.

NOTE: The chapters in this book, especially Chapter 10, frequently use the term "file-identifer." Other books that describe Pascal may just call this identifier "file." The file-identifier is not the same thing as a file specification. The file specification identifies the actual AMOS disk file that you want to read data from or write data to. The file-identifier identifies the Pascal file variable. Think of the file-identifier as specifying the Pascal data structure with which the actual file will be associated. Several of the functions you can use to handle files accept a file-identifier and a file specification. For example, the FSPEC procedure accepts three arguments: the file-identifier, an AMOS filespec, and a default extension. For instance:

```
FSPEC(File1,'ACCNTS,''DAT');
```

where File1 is the file-identifier, and ACCNTS.DAT is the AMOS file we want to associated with that file variable.)

The standard identifier TEXT has been pre-declared for you; this identifier is equivalent to FILE OF CHAR. (See Section 7.2.4, above, for information on TEXT.)

7.2.7 RECORD

A record is a data structure that consists of a number of components (called "fields"). Unlike arrays, the record elements do not have to be of the same type, and you access the elements by name, not by subscript. You can use records to develop very sophisticated data structures (e.g., array of records, file of records, pointers to records).

When you declare a record type, you are defining a template for a group of variables that contain related information, but which do not have to be of the same type. To define a record, use the type declaration. You will provide the name of the record, and names and types of the fields within that record:

```
TYPE Identifier =
            RECORD  field-name...,field-nameN : field-type1;
                    field-name...,field-nameN : field-type2;
                                    .
                                    .
                                    .
                    field-name...,field-nameN : field-typeN;
              END;
```

For example, a record to represent a date could be defined as:

```
TYPE  Date =
          RECORD
            Month : (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sept,Oct,Nov,Dec);
            Day   : 1..31;
            Year  : INTEGER
          END;
```

You may then declare a variable of type Date:

```
VAR Deadline : Date;
```

Such a variable would contain three pieces of information: the month, the day, and the year. However, all the information may be treated as a unit if you want to do so.

If several fields share the same type, you may list them on one line, separated by commas. You may also nest record definitions. For example:

```
TYPE  Credit =
          RECORD
             Finances : RECORD
                           Checking, Savings, Loans : INTEGER;
                        END;
             Name     : STRING[50];
             Birth    : Date
          END;
```

After defining a record, you may then declare a variable of that type. For example:

```
VAR   Customer : Credit;
```

To select a field of a record, use both the name of the record variable and the name of the field, separated by a period. For example:

```
IF Customer.Name = 'Smith, John C.' THEN CheckCredit;
```

You may assign the value of record to another. For example, given:

```
VAR   Customer, Employee : Credit;
```

you may assign the contents of record Customer to record Employee:

```
Employee := Customer
```

which is equivalent to:

```
Employee.Finances := Customer.Finances;
Employee.Name := Customer.Name;
Employee.Birth := Customer.Birth;
```

7.2.7.1 Variant Parts - Records of the same type do not necessarily have to contain the same fields. Suppose, for example, that you are maintaining a record of customer information in which one of the fields tells you whether or not the customer has a car.

```
Car : Boolean;
```

If, in fact, the customer does have a car, you might want to maintain another set of information (such as license number, model, year of make, etc.), but it doesn't make sense to fill in that information for a customer who doesn't have a car. Pascal allows you to allocate fields which may or may not exist, depending on the value of another field. These fields, which act as variations to the basic record structure, are called "variant" fields. The variant field definition takes this form:

```
CASE field-type OF
          Case-label...,Case-labelN : (field-list1);
          Case-label...,Case-labelN : (field-list2);
                        .
                        .
                        .
          Case-label...,Case-labelN : (field-listN)
```

or:

```
CASE case-field-identifier : field-type OF
            Case-label...,Case-labelN : (field-list1);
            Case-label...,Case-labelN : (field-list2);
                          .
                          .
                          .
            Case-label...,Case-labelN : (field-listN)
```

Several case labels may be written on one line, separated by commas. The list of variant fields must be enclosed with parentheses. (If no variant fields are to be used in the case of a certain value, empty parentheses may be used or the value may be omitted.) If you create a variant part, the variant fields must appear at the end of the record definition. For example:

```
TYPE  Customer =   RECORD
                       Name : STRING[50];
                    Number : INTEGER;
                    CASE Car : BOOLEAN OF
                         TRUE : (LicenseNo : STRING[7];
                                 Model : STRING[15];
                                  Year : INTEGER);
                        FALSE : ()   { You may omit this line }
                END;

VAR   Query : ARRAY [1..200] OF Customer;
```

7.2.8 Pointer Type

Pascal recognizes two categories of variables: static and dynamic.

Static Variables - Static variables are declared in variable declarations which determine their types and identifiers. You use these identifiers to refer to the variables. Static variables are created when the block in which they are declared is executed, and remain in effect until your program leaves that block. Most of the variables shown in this book are static variables. They can only be used when you know ahead of time what the storage requirements of your program is going to be.

Dynamic Variables - Dynamic variables are created on demand. They do not appear in variable declarations, and so cannot be referenced by variable identifiers. Instead, each dynamic variable of type X has associated with it a value of type ^X which is called the pointer to X. The pointer to X is used to access the corresponding dynamic variable, and contains the value of the address of the value.

The pointer type is declared via the type declaration statement:

```
TYPE  Identifier = ^base-type;
```

(The ^ symbol identifies a pointer.) For example:

```
TYPE  Location = ^INTEGER;
```

The declaration above establishes a pointer type Location whose pointer variables will point to variables of type RECORD. To use the pointer type, we must declare variables:

```
VAR   NewNumber : Location;
```

NewNumber is a pointer variable that is associated with an integer value. An identifier followed by the pointer symbol, ^, designates the actual value being pointed to. Therefore, NewNumber^ is the actual integer value being pointed to by NewNumber.

Now, to actually use the data types we have defined, we must use the NEW function to allocate the dynamic variable:

```
NEW(NewNumber);
```

creates an unnamed variable of type INTEGER, and stores the pointer to it in NewNumber. To access the new pointer, we reference it as NewNumber^. (See Section 11.1.4, "NEW," for information on NEW. AlphaPascal also uses two functions called MARK and RELEASE for manipulating pointer data; see Sections 11.1.3, "MARK," and 11.1.7, "RELEASE.)

Pascal contains a special pointer constant that indicates that a pointer is not pointing to anything: NIL. This is useful for indicating special conditions, such as the end of a list. For example:

```
EndingNode := NIL;
```

The use of pointers gives the Pascal programmer an extremely powerful tool for developing sophisticated structures (for example, linked lists). There are many examples of useful applications for pointers. As one simple example, suppose you want to sort an array of records:

```
TYPE  Rec = RECORD
              Name : STRING;
              Data : ARRAY [1..50] OF INTEGER
            END;

VAR     X : ARRAY [1..20] OF Rec;
```

you would have to perform a great many record moves; a slow and inefficient process. If you instead use pointers:

```
VAR X : ARRAY [1..20] OF ^Rec;
```

you only need to sort pointers, which is much faster. Here is a very small sample of the use of pointers:

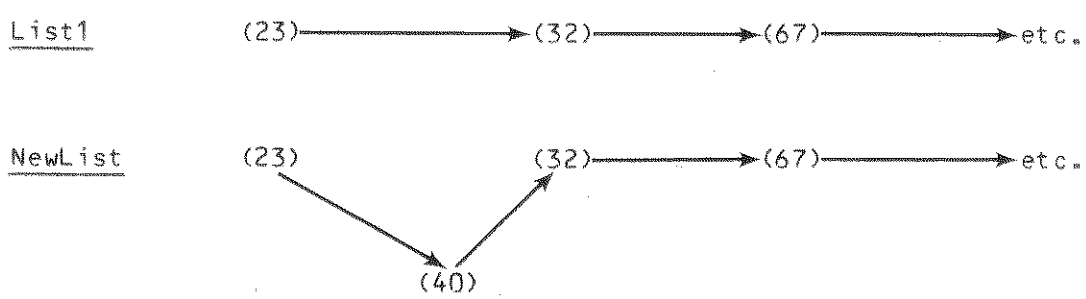
```
VAR   X, Y : ^INTEGER;

BEGIN
  NEW(X);
  Y := X;
  X^ := 5;  WRITE(X^);
  Y^ := 6;  WRITE(Y^);
  WRITE(X^)
   { Note, X and Y are pointing to the same
     location, so output will be 5,6,6 }
END.
```

A linked list is one example of a useful data structure you can build with pointers. (You might also consider building doubly linked lists, trees, queues, etc.) Let's take a look at the linked list and see why it is so useful, and how to build one.

Each element of a linked list contains: 1) data; and 2) a pointer to the next element of the list. To change the order of the elements in the list, therefore, you only have to change the pointers, not the elements themselves.

Let's say that you have a sorted array of integers. If you add another number to the array, you must sort the entire array to get the elements back into the proper order. If, however, the numbers are stored as a linked list, adding a new number just entails changing two pointers in the list. For example:



To delete an element of the list, you only need to link around it.

Declare a linked list as follows:

```
TYPE   Node = RECORD
                 Data : INTEGER;
                 Next : ^Node
               END;
```

Notice that we said that the data portion of the list element will hold integer data; you can use whatever data type you want.

Let's build a simple linked list, and then display it in reverse:

```
PROGRAM LinkedList;

TYPE  Pointer = ^Element;

        Element = RECORD
                    Data : INTEGER;
                    Next : Pointer
                  END;

VAR   I,X : INTEGER;
        P,List : Pointer;

BEGIN { LinkedList }
  WRITE('Enter integer: ');     { Get first number of list }
  READLN(X);
  List := NIL;          { Initialize list }
  WHILE X <> 0 DO { End list when X = 0 }
    BEGIN
      NEW(P);             { Allocate dynamic variable }
      P^.Data := X;       { Put number into list }
      P^.Next := List;  { Set list pointer to next element }
      List := P;
      WRITE('Enter integer: ');
      READLN(X)
    END;
  P := List;
  WHILE P <> NIL DO
    BEGIN
      WRITELN(P^.Data);
      P := P^.Next
    END
END { LinkedList }.
```

If you enter the numbers: 1 2 3 4 5 6 7, you see displayed: 7 6 5 4 3 2 1. Other useful examples would involve inserting elements into a list and deleting elements from a list by updating the list pointers.

NOTE: AlphaPascal contains the procedures MARK and RELEASE which you use in combination with NEW to make use of a stack-like structure called the "heap." (See Chapter 11 for information on MARK and RELEASE.) MARK and RELEASE allow you to perform very powerful operations with dynamic variables. However, they can be dangerous if used unwisely; you should be an experienced Pascal programmer before using MARK and RELEASE.

CHAPTER 8

EXPRESSIONS

An expression is any combination of operators, constants, function calls, and variables. For example:

```
(238.6 * Invoice + SQRT(TaxBill))/365
```

This chapter discusses the legal AlphaPascal operators, and gives the rules of operator precedence. We also talk about some special expression handling abilities of AlphaPascal.

8.1 OPERATORS

An operator is a symbol that directs Pascal to perform an action on the elements of an expression. For example, the addition operator, +, in the expression 34+123 tells Pascal to add the numbers 34 and 123. The operator types in Pascal are: arithmetic, Boolean, relational, logical, and set. Another special operator, the assignment operator, is used to assign values to variables.

8.1.1 Operator Precedence

When Pascal sees the various operators in an expression, it evaluates the elements in the expression in response to those operators. When more than one type of operator appears in one expression, Pascal follows a set of rules called "operator precedence" in determining which operators to act upon first. If the precedence of all operators in the expression is the same, Pascal evaluates the expression from left to right. For example, Pascal evaluates the expression:

```
312 + 34 - 20
```

as:

```
(312 + 34) - 20
```

evaluating the value 312+34 first, and then subtracting 20 from it. If the precedence of the operators differs, Pascal evaluates the elements connected by the operator of highest precedence first, and then evaluates the elements connected by the operator of the next highest precedence, and so on. For example, multiplication has a higher precedence than addition, so the expression:

```
76 * 54 + 2
```

tells Pascal to multiply 76 by 54, and then add 2 to that value. The expression evaluates to (76 * 54) + 2, or 4106.

You can change the order in which Pascal processes operators by using parentheses. Pascal always evaluates elements in the innermost set of parentheses first, and then works outward. For example, if you want Pascal to act upon the addition operator first in the previous example, you must use parentheses to tell Pascal to apply that operator first:

```
76 * (54 + 2)
```

This expression tells Pascal to add 54 and 2, and then multiply that value by 76. The expression thus evaluates to 4256.

NOTE: The operator precedence used by AlphaPascal differs slightly from that used by standard Pascal. We have changed the precedence to be compatible with that of other language processors on the Alpha Micro system. Specifically, in AlphaPascal the Boolean operators are of lower precedence than the relational operators. The only time you will need to worry about this is if you use expressions that compare unparenthesized Boolean expressions with relational operators (e.g., NOT A = B).

If your programs must be written to be compatible with standard Pascal (for instance, if you want to be able to transfer your programs to another computer system that uses standard Pascal) use parenthese to make sure that your expressions are evaluated in accord with standard Pascal's rules of operator precedence. For example, the expression:

```
NOT A = B
```

is evaluated by AlphaPascal as: NOT (A = B).

If you want the expression to work for either standard Pascal or AlphaPascal, you should either write it as:

```
(NOT A) = B
```

or:

```
NOT (A = B)
```

to indicate how you wish the expression to be evaluated.

The table below gives the rules of operator precedence for AlphaPascal:

Highest Precedence

Parenthesized expressions	
Sign inversion:	- (unary)
Multiplying operators:	* / DIV MOD
Adding operators:	+ -
Relational operators:	= <> < > <= >= IN
Boolean operators:	NOT AND OR

Lowest Precedence

8.1.2 Assignment Operator

The assignment operator, :=, assigns the value of an expression to a variable. (See Section 9.1, "Assignment Statement," for information on its use in a program statement.)

Pascal evaluates the expression on the right side of the assignment operator symbol, :=. The variable on the left side of the assignment operator then assumes the value of that expression. Note that all variables to which values are assigned must have been previously declared. For example:

```
CardValue := 9.56
```

assigns the value 9.56 to the variable CardValue. The expression above must have been preceded in the program by a statement such as:

```
VAR CardValue : REAL
```

which declares that the variable CardValue may only assume real number values.

Most languages (including standard Pascal) only allow the value of a variable to be changed by an assignment statement. Alphapascal allows the value of a variable to be changed within an expression. For example:

```
200 + Sum/Total := 365
```

Pascal reads the expression above as:

```
(200 + (Sum/(Total := 365))
```

That is, Pascal assigns the value 365 to the variable Total, and then divides the value Sum by Total (which is now 365), and adds 200 to it.

Remember that the assignment operator has the highest precedence, and that Pascal evaluates expressions from left to right when operator precedence is equal. The Assignment operator has extremely high "left precedence," and very low "right precedence." That means that it "binds" itself strongly to the nearest element on the left, but loosely to the remaining elements on the right. To make this idea clearer, consider the following expression:

```
Result := 10 + Score - Cards := 32 + Pairs - Singles
```

The second assignment operator binds strongly to the variable Cards, but "swallows up" all of the expression to the right of itself. This means that AlphaPascal evaluates the expression above as:

```
Result := (10 + Score - (Cards := (32 + Pairs) - Singles))
```

That is, Cards is set to (32 + Pairs) minus the value of Singles. Then, the value of Cards is subtracted from 10 + Score. That value is assigned to the variable Result.

As another example of the use of the assignment operator in an expression, consider a situation where you want to initialize a group of variables by setting their values to zero. Pascal does not have a multiple assignment statement. However, the expression:

```
Averages := Total := Sum := Median := 0
```

causes Pascal to perform a multiple assignment as a side effect of evaluating the expression.

8.1.2.1 Modifying Assignment Operators - AlphaPascal contains a set of special operators called "modifying assignment operators." These operators allow you to assign values to variables by modifying the value of the variable instead of replacing that value. For example, the assignment expression:

```
RecordCount := 120
```

tells Pascal to replace the value of RecordCount with the number 120. A modifying assignment expression of the form:

```
RecordCount += 120
```

tells Pascal to take the value of RecordCount and modify it by adding 120 to it. Pascal then assigns this new value to RecordCount. We thus modify, rather than replace, the value of RecordCount. In effect, the expression above is equivalent to:

```
RecordCount := RecordCount + 120
```

The modifying assignment operators are:

+=	Adding modifying assignment operator
-=	Subtracting modifying assignment operator
*=	Multiplying modifying assignment operator
/=	Dividing modifying assignment operator

As another example, the statements:

```
Number := 1;
  FOR I = 1 TO 5 DO
    Number *= 2 {Same as 'Number := Number * 2'}
```

compute two to the fifth power. So, Number takes on the values 2, 4, 8, 16 and 32.

8.1.3 Arithmetic Operators

The arithmetic operators are:

+ (unary)	Identity	Takes INTEGER or REAL operands; result is same type as operands.
- (unary)	Sign inversion	Takes INTEGER or REAL operands; result is same type as operands.
+	Addition	Takes INTEGER or REAL operands; result is same type as operands.
-	Subtraction	Takes INTEGER or REAL operands; result is same type as operands.
*	Multi-plication	Takes INTEGER or REAL operands; result is same type as operands.
DIV	Integer division	Takes INTEGER operands; result is INTEGER.
/	Real division	Takes INTEGER or REAL operands; result is INGEGER or REAL.
MOD	Modulus	Takes INTEGER operands; result is INTEGER.

NOTE: If you wish to use the sign inversion symbol, -, you must enclose the number in parentheses if another operator precedes the number. For example, the expression 3 * -5 is illegal, but the expression:

```
3 * (-5)
```

is valid, and evaluates to -15.

8.1.4 Relational Operators

=	Equality	Scalar, STRING, SET, or pointer operands. BOOLEAN result.
<>	Inequality	Scalar, STRING, SET, or pointer operands; BOOLEAN result.
<	Less than	Scalar or STRING operands; BOOLEAN result.
>	Greater than	Scalar or STRING operands; BOOLEAN result.
<=	Less than or equal	Scalar or STRING operands; BOOLEAN result.
	(or set inclusion (subset))	SET operands; BOOLEAN result.
>=	Greater than or equal	Scalar or STRING operands; BOOLEAN result.
	(or set inclusion (superset))	SET operands; BOOLEAN result.
IN	Set membership	First operand is any scalar, second is its SET type. BOOLEAN result.

8.1.5 Logical Operators

NOT	Negation	BOOLEAN operands; BOOLEAN result.
AND	Conjunction	BOOLEAN operands; BOOLEAN result.
OR	Disjunction	BOOLEAN operands; BOOLEAN result.

8.1.6 Set Operators

+	Union	Given sets of type X, result is of type X.
-	Set difference	Given sets of type X, result is of type X.
*	Intersection	Given sets of type X, result is of type X.

8.2 CONSTANTS

A constant is a value that doesn't change. For example, the number 34.5 is a constant, because it can assume no other value. Certain constants have been pre-defined by Pascal for your use. They are:

```
MAXINT  the maximum integer AlphaPascal can represent.
FALSE   Boolean false
TRUE    Boolean true
```

You can use these constants as you would any others. For example:

```
{ Find the minimum of a list of numbers.  Initialize CurrentMin to
  largest possible number. }

CurrentMin := MAXINT;
REPEAT
  READ(DataFile,NewNumber);
  IF NewNumber < CurrentMin THEN CurrentMin := NewNumber;
UNTIL EOF     { Continue till end of file is reached };
WRITELN('Smallest number is: ',CurrentMin);
```

Pascal allows you to assign a name to a constant so that you can identify it by name within a program, rather than including the constant itself. For example, it would be rather cumbersome if you had to include the numeric constant 3.14159 throughout a program. Once you use a constant definition statement to assign 3.14159 a name (such as Pi), you can refer to that constant by name. For example:

```
WRITELN('The Circumference = ',Pi * 234);
```

You may also assign a name to a string constant. For information on naming constants, see Section 6.3, "Constant Definitions." For information on the form string and numeric constants may take, see Section 5.6, "Notation." NOTE: Of course, constants are not variables; that is, you may not assign a constant a new value within the program block.

8.3 VARIABLES

A variable is a named symbol that represents a value. For example, the variable named StudentID might assume a range of student identification numbers. Variables allow a program to operate on a variety of data.

Each variable in a program may assume only one type of value (e.g., integer values, real values, Boolean values, etc.). Pascal requires that you declare the data type of each variable before that variable is used. (See Section 6.5, "Variable Declarations," and Chapter 7, "Data Types." for information on data types and declaring variables.)

For information on choosing a valid name for a variable, see Section 5.4, "Legal Identifiers." A variable identifer may be in the form of an expression. For example, consider the case where we want to refer to an element in an array:

```
NewArray[2,4] := 99;
```

8.4 IF-THEN-ELSE EXPRESSIONS

Wherever an expression is legal, AlphaPascal allows you to include an IF-THEN-ELSE expression. This allows you to conditionally evaluate one of two alternative expressions. The construct takes the form:

```
IF condition THEN expression ELSE expression
```

Note that you must include the ELSE clause if you use the IF-THEN construct in this way. For example:

```
IF Credit > (IF BillAmt > 1000 THEN 2000 ELSE 0)
  THEN WRITELN('OK, charge it.')
  ELSE WRITELN('Sorry, send it C.O.D.');
```

The statement above contains this expression: IF BillAmt > 1000 THEN 2000 ELSE 0. This evaluates either to 2000 or to 0, depending on whether or not the variable BillAmt has a value greater than 1000. Therefore the statement above either evaluates to:

```
IF Credit > 2000
  THEN WRITELN('OK, charge it.')
  ELSE WRITELN('Sorry, send it C.O.D.');
```

or:

```
IF Credit > 0
  THEN WRITELN('OK, charge it.')
  ELSE WRITELN ('Sorry, send it C.O.D.');
```

Remember that expressions can also contain string constants or variables. Consider the following small program that conditionally assigns a value to ErrorReport:

```
PROGRAM Recovery;

VAR       ErrorFlag : BOOLEAN;
        ErrorReport : STRING;

BEGIN { Recovery }
   ErrorFlag := FALSE;
   ErrorReport := (IF ErrorFlag
      THEN 'An error occurred!' ELSE 'No error.');
   WRITELN(ErrorReport)
END { Recovery }.
```

NOTE: Including an IF-THEN-ELSE construct in an expression is not a feature of standard Pascal. Note that IF-THEN-ELSE may not be used in a variable expression. For example:

```
(IF X THEN Y ELSE Z) := 1
```

is illegal.

8.5 CASE EXPRESSIONS

Wherever an expression may appear, AlphaPascal allows you to include a CASE expression. This allows you to conditionally evaluate one of several alternative expressions. (NOTE: This is not a feature of standard Pascal.) The expression must take the form:

```
CASE value OF
        value1 : expression;
        value2 : expression;
                 .
                 .
         ELSE expression
```

For example:

```
WRITE(CASE ErrorCode OF
             1 : 'Illegal input';
             2 : 'Number too large';
             3 : 'Number too small';
             ELSE 'undefined error');
```

The statement above chooses one string to write, depending on the value of the variable ErrorCode. For example, if ErrorCode contains a value of 3, the statement above evaluates to:

```
WRITE('Number too small');
```

If ErrorCode contains a value that is not 1, 2, or 3, the statement evaluates to:

```
WRITE('Undefined error');
```

CHAPTER 9

STATEMENTS

9.1 ASSIGNMENT STATEMENT

The assignment statement assigns a value to a variable. It takes this form:

```
variable := expression
```

Pascal evaluates the expression on the right side of the assignment operator symbol, :=. The variable on the left side of the assignment operator then assumes the value of that expression. Note that all variables to which values are assigned must have been previously declared.

For example, given that your program previously contained the statement:

```
VAR   AccountNum : INTEGER;
```

the statement:

```
AccountNum := 1024+1
```

assigns the integer value 1025 to the variable AccountNum. For more information on the assignment operator, see Section 8.2, "Assignment Operator." That section also discusses the use of the assignment operator in expressions, discusses the precedence of the assignment operator, and describes the AlphaPascal modifying assignment operators.

9.2 PROCEDURE CALLS

Procedure invocations may appear as program statements. (For information on procedure parameters, see Section 6.6.1, "Formal Parameters.") Liberal use of procedure calls in your programs illustrates one of the important features of Pascal-- modularity. Given the appropriate procedure definitions, a main program can be extremely easy to read. For example:

```
BEGIN { Main Inventory }
   OpenFiles(Receivng,Manufact) { Input filespecs for data from Receiving,
                                  Manufacturing departments.  Open files.}

   ReadData(Receivng,Manufact)  { Read inventory parts lists };
   FindLow(LowFile)             { Compute which parts we are low on
                                  and write to file. };

   PrintReport(Date,LowFile)    { Print list of parts we need more of}
END { Main Inventory }.
```

We can tell just by looking approximately what the program does. The procedures OpenFiles, ReadData, FindLow, and PrintReport do the actual work.

9.3 EXIT

EXIT allows you to exit from the program to the monitor or from a procedure or function to a calling program or routine. EXIT takes one argument-- the keyword PROGRAM or the name of the procedure or function you want to exit from. For example:

```
EXIT(PROGRAM);

EXIT(EvalErr);
```

(You may not supply EXIT with the program identifier; use the PROGRAM keyword to exit a program.)

9.4 GOTO STATEMENT

The GOTO statement takes the form:

```
GOTO label;
```

where "label" has previously been defined in a label declaration statement. The label may not lie out of the current procedure or function block. For example:

```
{$G+}
PROGRAM Tip;

VAR     Cost, Percent, Tip : REAL;
                 Query : CHAR;

LABEL 100;

      BEGIN {Program Tip}
         WRITELN('Let''s calculate the waiter''s tip');
         WRITE('Was it good service (Y or N)?: ');
         READLN(Query);
         IF Query = 'N' THEN GOTO 100;
           WRITE('How much did you pay for dinner? ');
           READLN(Cost);
           WRITE('What percentage do you want to tip? ');
           READLN(Percent);
           Percent *= 0.01;
           Tip := Percent * Cost;
           WRITELN('The tip is: ',Tip);
100:  END {Program Tip}.
```

NOTE: The AlphaPascal compiler is initially set so that it does not recognize GOTO statements; that is, it gives the error message "Illegal symbol" if it encounters a GOTO statement in your program. To tell the compiler that you want to use GOTO statements in a particular program, the compiler option $G+ must appear at the front of that program. (For information on the $G compiler option, see Section 4.3.2.1, "The GOTO Options ($G+ and $G-).")

9.5 NULL STATEMENT

One of the features that make Pascal programs especially flexible is the fact that you may include a null statement within your programs. A null statement allows you to include extra semicolons within compound statements, and to omit statements in certain program constructs. For example, consider the CASE expression below:

```
CASE expression OF
        1 : statement1;
        2 : statement2;
        3 : ;              { Null statement }
        4 : statement3;
      ELSE statement4
```

By including just a semicolon after value 3, we tell the CASE expression to perform no statement if the expression evaluates to 3.

As another example:

```
IF A = B THEN
   IF C = D THEN Flag := TRUE
      ELSE                    { Null statement after ELSE }
   ELSE NewFlag := True;
```

The use of the null statement above allows us to attach the second ELSE to the first IF-THEN construct. (Otherwise, the second else would be performed when C <> D, rather than when A <> B.)

9.6 COMPOUND STATEMENT

The body of a Pascal program is a compound statement; that is, it is marked with the BEGIN and END keywords, and contains one or more statements between those keywords (even if the enclosed statement(s) is a null statement-- see the paragraph above, Section 9.5, "The Null Statement").

Each individual statement may also consist of a compound statement. The use of compound statements is what gives a Pascal program its nested, block structure. Many sample programs in this book contain several BEGIN-END blocks.

(See Section 6.2, "Label Declarations," for information on labeling BEGIN-END keyword pairs. Labeling these keywords tells the compiler to report back to you with an error message if the BEGIN-END keywords are not matched as your labels have indicated they should be.)

9.7 CONDITIONAL STATEMENTS

Conditional statements allow you to execute certain sections of code only if specific conditions are satisfied. This section discusses the IF-THEN, IF-THEN-ELSE, CASE-OF, and CASE-OF-ELSE statements.

9.7.1 IF-THEN

The IF-THEN statement takes the form:

```
IF Boolean expression THEN statement;
```

where statement may, of course, consist of a compound statement. A Boolean expression is one which evaluates to a Boolean value. For example: 1>5 is evaluated as FALSE, since 1 is not greater than 5. For example:

```
IF TestScore > 90 THEN WRITELN('Congratulations! An A+');
```

The statement(s) following the THEN clause are carried out if the Boolean expression evaluates to TRUE; if it evaluates to FALSE, control is transferred to the next statement after the IF-THEN statement.

Note that the statement following the THEN keyword may itself be an IF-THEN statement. For example:

```
IF Single THEN
    IF Withholding > .36 THEN Dependents := 1;
```

(Which is the same as: IF Single AND (Witholding > .36) THEN...) If Single evaluates to TRUE, everything after the first THEN keyword is executed; otherwise, control passes to the next program statement.

NOTE: You may include the keywords IF-THEN in an expression to conditionally evaluate one of two alternative expressions. See Section 8.4, "IF-THEN-ELSE Expressions."

9.7.1.1 IF-THEN-ELSE - The addition of an ELSE clause to an IF-THEN statement gives us a way to select one of two statements as a result of evaluating an expression. The IF-THEN-ELSE statement takes the form:

```
IF Boolean expression THEN statement-1 ELSE statement-2;
```

If the Boolean expression is TRUE, the first statement is executed; otherwise, the second statement is executed. As in the case of the simple IF-THEN statement above, a compound statement may appear in place of a single statement. One of the two statements will always be executed. For example:

```
IF Margin > LineWidth THEN Error := PGWDTH ELSE LineWidth -= Margin;
```

The line above is from a program that formats documents. If the value for Margin is greater than the current LineWidth, then we set an error code into the Error flag; otherwise, we reset the LineWidth to the old value minus the Margin.

What happens if an IF-THEN-ELSE statement contains multiple IF-THEN statements? To which IF-THEN statement does the ELSE apply? For example:

```
IF A = B THEN IF B = C THEN Flag := 0 ELSE Flag := 1;
```

Does Flag get set to 1 if A>B or if B>C? AlphaPascal nests ELSEs. That means that in the case above, the ELSE applies to the last IF-THEN statement; if B=C is FALSE, Flag is set to 1. As another example:

```
PROGRAM DoubleElse;

VAR   A,B,C,D : REAL;

BEGIN { DoubleElse }
  WRITE('Enter A, B, C, D: ');
  READLN(A,B,C,D) { Enter values for A,B,C,D };

  IF A = B
    THEN IF C = D
      THEN WRITELN('No Else')
    ELSE WRITELN('Else1')
  ELSE WRITELN('Else2')

END { DoubleElse }.
```

As we said, ELSEs are nested. That means that the second ELSE is applied if the first IF clause (A=B) is false; the first ELSE is applied if the second IF clause (C=D) is false. So, the output from the program above is as follows:

A=B	C=D	Output
True	True	No Else
False	True	Else2
True	False	Else1
False	False	Else2

9.7.2 CASE-OF

The CASE statement allows you to select one out of a group of statements for execution. The CASE statement takes this form:

```
CASE expression OF
    Case-label...,Case-label : statement1;
    Case-label...,Case-label : statement2;
                        .
                        .
                        .
    Case-label...,Case-label : statementN
  END
```

The expression (called the "selector") is evaluated, and its value must be the same as one of the case-labels. A selector must not be of type REAL, and it must be of the same type as the case-labels. You may have as many case-labels as you like, but each case-label may appear only once in any one CASE statement. When a matching case-label is found, the statement following that case-label is executed. For example:

```
BEGIN { MainMenu }
  WRITE('Enter your choice from the menu above :');
  READLN(MenuChoice);
  CASE MenuChoice OF
    'A' : ComputeTax;
    'B' : UpdateAccount;
    'C' : PrintReport;
    'D' : DoBilling
  END { End of CASE };
END { MainMenu }.
```

The program block above performs the proper procedure based on the user selection from the main menu.

NOTE: What happens if none of the case-labels match the selector? Standard Pascal says that such an event is undefined. AlphaPascal simply says that if none of the case-labels are matched, then control passes to the next program statement. (See the next paragraph for information on using an ELSE clause to catch a situation where no match occurs.)

9.7.2.1 CASE-OF-ELSE - AlphaPascal allows a unique variant to the CASE statement; the CASE-OF-ELSE statement. This statement takes the form:

```
CASE expression OF
    Case-label...,Case-label : statement1;
    Case-label...,Case-label : statement2;
                        .
                        .
                        .
    Case-label...,Case-label : statementN
  ELSE statement;
```

For example:

```
BEGIN { MainMenu }
  WRITE('Enter your choice: ');
  READLN(MenuChoice);
  CASE MenuChoice OF
    'A' : ComputeTax;
    'B' : UpdateAccount;
    'C' : PrintReport;
    'D' : DoBilling
  ELSE WRITELN('No valid choice') { Didn't enter A,B,C, or D };
END { MainMenu }.
```

Notice that the ELSE clause takes the place of the final CASE statement END keyword.

NOTE: See Section 8.5, "CASE Expressions," for information on using the CASE construct to conditionally evaluate one of several alternative expressions.

9.8 REPETITIVE STATEMENTS

It is often the case that one section of a program must be performed repetitively, based on a certain condition. AlphaPascal provides a number of repetitive statements: WHILE-DO, REPEAT-UNTIL, and FOR-DO. It is important that you decide which of these statements is exactly correct for your application, since each differs somewhat in the way that it handles final values.

9.8.1 WHILE-DO

The WHILE-DO statement takes the form:

WHILE Boolean expression DO statement

where the Boolean expression evaluates to a TRUE or FALSE, and the statement may consist of a compound statement. For example:

```
PROGRAM;

VAR   Counter, Number, Average, Sum : REAL;

BEGIN { Main Program }
  Number := 1              { Initialize Number to > 0. }
  Average := Counter := 0;
  WHILE Number > 0 DO
    BEGIN
      WRITELN('Average: ',Average);
      Counter += 1;
      WRITE('Enter number: ');
      READLN(Number);
      Sum += Number;
      Average := Sum/Counter;
    END;
END { Main Program }.
```

In effect, you tell Pascal, "While the following condition is TRUE, execute the following statements." As soon as the condition becomes FALSE, the program finishes executing the entire WHILE loop, and then goes on to the next program statement. It is possible that a WHILE loop will never be executed if the initial condition is not true and never becomes true.

9.8.2 REPEAT-UNTIL

The REPEAT-UNTIL statement takes this form:

REPEAT statement-list UNTIL Boolean expression

where statement-list may be series of statements separated by semicolons, and expression evaluates to TRUE or FALSE. For example:

```
PROGRAM;

VAR   Number : INTEGER;
      Error : BOOLEAN;

BEGIN { Main program }
   Error := FALSE;
   REPEAT
        WRITE('Enter an integer divisible by 3: ');
        READLN(Number);
        IF (Number MOD 3) = 0 THEN
             WRITELN('Correct. Try another.') ELSE Error := TRUE
   UNTIL Error
   WRITELN('Incorrect. End of exercise.')
END { Main Program }.
```

Because the REPEAT-UNTIL keywords appear at the beginning and end of the loop (making it clear where the beginning and end of the loop are), we do not have to include the BEGIN-END keywords after the REPEAT keyword (however, you may do so if you wish). A REPEAT loop will always be executed at least once.

9.8.3 FOR-DO

The FOR-DO statement allows you to execute a given statement or group of statements a specific number of times. A FOR-DO loop is executed for every value of the "control variable" from some starting value up to and including some terminal value. A control variable must not be of type REAL. The FOR-DO statement takes this form:

```
FOR Variable-identifier := expression TO expression DO statement
```

For example:

```
PROGRAM;

VAR   Counter : INTEGER;

BEGIN { Main Program }
  WRITELN('The square roots of the integers 1 to 10 are :');
  WRITELN;
  FOR Counter := 1 TO 10 DO WRITELN('Square root: ',SQRT(Counter))
END { Main Program }.
```

Each time the statement after the DO keyword is executed, Counter is incremented by one. The program above prints the square roots of the integers from 1 to 10.

A variant of the FOR-DO loop exists that allows you to decrement the control variable. It takes the form:

```
FOR Variable-identifier := expression DOWNTO expression DO statement
```

Each time the statement after the DO keyword is executed, the control variable is decremented by one. Note that it is possible that a FOR-DO loop may not be executed at all, if the initial and terminal values of the control variable are not in the proper range. (For example, the statement FOR I := 5 TO 1... will not be executed, but FOR I := 5 DOWNTO 1... will be executed.)

9.9 WITH-DO

The WITH-DO statement allows you to access fields of a record as if they were simple variables. The WITH-DO statement takes the form:

```
WITH Variable-identifier1...,Variable-identifierN DO statement
```

The WITH-DO statement simply gives you a shorthand way of accessing record fields without specifying the name of the record structure for each access. (See Section 7.2.7, "RECORDS," for information on records.) For example, suppose you have a record made up of the following fields:

```
CarInfo.Model
CarInfo.Year
CarInfo.Color
CarInfo.SerialNumber
```

You have 100 cars on your car lot, and you want to know how many of them are red. The records may be set up this way:

```
TYPE   CarInfo = RECORD
                    Model : STRING[3];
                    Year  : INTEGER;
                    Color : STRING[3];
             SerialNumber : INTEGER;
                  END { record };


VAR      Counter,CarNumber : INTEGER;
                   CarLot : ARRAY [1..100] OF CarInfo;
```

Now you can process them. Without using a WITH-DO statement, you would have to do something like this:

```
Counter := 0;
FOR CarNumber := 1 TO 100 DO
  BEGIN
    IF (CarLot[CarNumber].Model='X20')
      AND (CarLot[CarNumber].Color='red')
        THEN Counter += 1;
    WRITELN('Number of red X20s is: ',Counter)
  END;
```

A more convenient way is to use the WITH-DO statement:

```
Counter := 0;
FOR CarNumber := 1 TO 100 DO
  BEGIN
    WITH CarLot[CarNumber] DO
      IF (Model='X20') AND (Color='red') THEN Counter += 1;
    WRITELN('Number of red X20s is: ',Counter)
  END;
```

By specifying more than one variable-identifier, you can use the WITH-DO statement to access fields that occur within record fields. For example, to access data in the record CarLot.Make.Model, you could write something like this:

```
WITH CarLot,Make DO
  Model := 'HatchBack';
```

This is equivalent to:

```
WITH CarLot DO
  WITH Make DO
    Model := 'HatchBack';
```

CHAPTER 10

INPUT/OUTPUT FUNCTIONS AND PROCEDURES

The functions and procedures discussed in this chapter are used to transfer data between your programs and the users of those programs, and between programs and files. The routines we describe in the first part of the chapter, "Basic Functions and Procedures," are routines that users of standard Pascal will probably be familiar with. The last part of the chapter, "Special Functions and Procedures for File I/O," contains descriptions of functions and procedures that are particularly for use with the AMOS file structure.

NOTE: You will notice that we use the term "file-identifier" when discussing a file variable, rather than the simple term "file" (sometimes used by other Pascal books). This is to help avoid confusing the file-identifier with the "file specification," which is the specification of the actual AMOS disk file that is associated with the file variable. Using an AMOS file requires that you first declare the file-identifier and then associate it with the file specification of an AMOS disk file. See Section 10.2, "Special Functions and Procedures for File I/O," for more information on using AMOS disk files, especially Section 10.2.12, "OPEN.")

10.1 BASIC FUNCTIONS AND PROCEDURES

These are the Input/Output functions and procedures that users of standard Pascal will be most familiar with. Later sections in this chapter discuss special input/output functions and procedures that allow your programs to access the AMOS file structure.

You will often use the procedures GET, PUT, READ, READLN, WRITE, and WRITELN for transferring data between your program and the users of your program. These procedures are also used to transfer data between your program and special storage areas called "files." The other procedures discussed in this section, PAGE, RESET, and REWRITE, are used only with files. Remember that when we talk about "files," we are referring to the special data type FILE that in AlphaPascal can be associated with AMOS disk files.

Three special pre-declared file-identifiers exist that you should be aware of:

INPUT Specifying INPUT tells AlphaPascal that you want to use the terminal as an input file. For example, when you use READLN to get data from the terminal keyboard:

```
READLN(EmployeeNumber,Dept);
```

you have implicitly said:

```
READLN(INPUT,EmployeeNumber,Dept);
```

(In other words, if you omit a file-identifier from the arguments given to the READLN procedure, READLN assumes you want to use INPUT.) INPUT is a TEXT file.

OUTPUT Specifying OUTPUT tells AlphaPascal to use the terminal as an output file. For example, when you write data to the terminal display via the WRITELN procedure:

```
WRITELN('Enter your Employee Number: ');
```

you have implicitly said:

```
WRITELN(OUTPUT,'Enter your Employee Number: ');
```

OUTPUT is a TEXT file.

KEYBOARD The KEYBOARD file-identifier acts much the same as INPUT, except that if the terminal is in Charmode, the characters typed by the user of your program will not echo on the terminal display. For example:

```
CHARMODE;
WRITELN('Enter password: ');
READ(KEYBOARD,Password);
```

Asks the user of your program for a password, but does not display the characters of the password as they are entered. When your terminal is not in Charmode and you are using INPUT, the monitor processes and filters your input. (For example, it appends a line-feed to the end of a carriage return.) KEYBOARD and Charmode give you a way to examine the input exactly as it is entered; the monitor does no processing of the characters. That means that for the example above to work, after typing the password, the user must type a carriage return AND a line-feed. KEYBOARD is a TEXT file. (For information on Charmode, see Section 11.2.1, "Charmode.")

INPUT, OUTPUT, and KEYBOARD are associated with the special AMOS file specifications TTY:, TTY:, and KBD:. See Section 10.2.1 for information on these special devices.

If you are using READ to input data, remember that you will have to do a READLN after end-of-line has been reached to make it read past the line-feed at the end of the carriage return in order to reset EOLN to FALSE. For example:

```
PROGRAM TestEOLN { Count how many characters are in input };

VAR     Source : CHAR { Input };
        Counter : INTEGER;

BEGIN { TestEOLN }
  WRITE('Enter a line of characters: ');
  READ(Source);
  Counter := 0;
  WHILE NOT EOLN DO
     BEGIN
       WRITE(Source);
       Counter += 1;
       READ(Source)
     END;
  WRITELN;
  WRITELN('-- number of characters = ',Counter);
  READLN { Restore EOLN }
END { TestEOLN }.
```

The program above keeps reading characters until the user enters a RETURN (that is, until EOLN is TRUE). Then it prints the number of characters in the input string. For example, a sample run of the program might look like this:

```
Enter a line of characters: NOW IS THE TIME
NOW IS THE TIME
-- number of characters = 15
```

10.1.4 GET and PUT

GET and PUT are the two basic file I/O procedures. You may use GET and PUT on files of any type, not just TEXT files.

10.1.4.1 GET

GET advances the buffer variable to the next file component. In doing so, it assigns the value of that file component to the buffer variable. The invocation takes the form:

```
GET(file-identifier);
```

where file-identifier is a file variable. If doing a GET moves the buffer variable past the end of the file, then the EOF function returns TRUE, and the contents of the buffer variable is undefined. So, save the contents of the buffer variable into another variable before doing a GET, if you need to access the very last item in the file.

10.1.4.2 PUT - PUT writes the value of the buffer variable into the component at the current file position. The procedure invocation takes the form:

```
PUT(file-identifier);
```

where file-identifier is a file variable. The EOF function remains TRUE.

10.1.4.3 Sample Program Using GET and PUT - Below is a very simple program using GET and PUT. Notice that we use the OPEN statement (described in Section 10.2.12) to associate the file-identifier DataFile with an AMOS disk file, NUMBER.DAT. The RESET procedure closes the file and re-opens it for input.

```
PROGRAM FileAccess;

VAR     DataFile : FILE OF CHAR;
           Entry : CHAR;
         Counter : INTEGER;

BEGIN { FileAccess }
  OPEN(DataFile,'NUMBER.DAT',OUTPUT);  { Open NUMBER.DAT for output }
  FOR Counter := 1 TO 5 DO
    BEGIN
      WRITE('Enter data: ');            { Get data from terminal }
      READLN(Entry);
      DataFile^ := Entry;               { Assign data to buffer var }
      PUT(DataFile)                     { Write to file }
    END;

  RESET(DataFile);                 { Close file and re-open for input }
  WHILE NOT EOF(DataFile) DO
  { Get data till file is empty }
    BEGIN
      Entry := DataFile^;
      WRITELN(Entry);
      GET(DataFile)                     { Get data from file }
    END;
END { FileAccess }.
```

NOTE: If you use OPEN to open a file for input, or if you use RESET, the first file component is placed into the buffer variable for you.

10.1.5 READ, READLN, WRITE, and WRITELN

The READ and READLN procedures are elaborations of the GET procedure (discussed above). You should use them only for TEXT files and terminal input. WRITE and WRITELN are elaborations of the PUT procedure (also discussed above); they are for use only with TEXT files and terminal output.

Although we say that these procedures are for use with TEXT files, you will notice throughout this book that we have made wide use of them for transferring data between programs and the terminal. Remember that your terminal is a TEXT file. Two TEXT files have been pre-declared for use with the terminal: INPUT and OUTPUT. If you omit the file-identifier from the list of arguments given to READ and READLN, the procedures assume that you want to use the file INPUT. No file-identifier in the list of arguments given to WRITE and WRITELN indicates that you want to use the file OUTPUT.

One last note on these procedures-- they convert REAL or INTEGER data to type CHAR. For example, when you say:

```
WRITE(Result);
```

where Result is an INTEGER variable containing the number 12, WRITE displays the characters "12" on your terminal. This is what you want to do when you send data to a terminal, but be careful in using READs and WRITEs on actual disk files. Consider performing file operations on a large file of INTEGER data. It would be very inefficient to handle that data in character form, since every time you manipulated it, you would have to re-convert it. It would be far better to use GETs and PUTs rather than READs and WRITEs to handle the numeric data, since GETs and PUTs do no conversion.

10.1.5.1 READ - The READ procedure inputs a list of variables from the terminal or a file. You should only use READ for TEXT files. NOTE: READ does not read an entire line of data up to a carriage return/line-feed.

Given the file variable Data, the procedure READ(Data,Character) performs these actions:

1. Scans over and ignores line-feed characters;
2. Character := Data^;
3. GET(Data);

The procedure invocation takes the form:

```
READ(file-identifier,list-of-variables);
```

If you omit the file-identifier:

```
READ(list-of-variables);
```

READ assumes that you want to use the file INPUT (that is, that you want to input from the terminal keyboard).

The READ arguments must be separated by commas. For example:

```
READ(DataFile,CustomerID,CustomerName);
```

where DataFile is a file variable, and CustomerID and CustomerName are variable identifiers. Or:

```
READ(Linesize,Pagesize,PageNumber);
```

where Linesize, Pagesize, and PageNumber are variables to be input from the terminal.

NOTE: If you input more than one variable via the READ or READLN procedure, those values should not be input separated by commas. For example, given:

```
READ(A,B,C);
```

The response:

```
1 2 3
```

is legal, but the response:

```
1,2,3
```

is not valid. If you respond with an illegal number (for example, you input an "A" for a variable of type INTEGER), AlphaPascal assigns a zero to that variable, instead of generating an error. It is the responsibility of your program to check the validity of data input by the READ procedure.

10.1.5.2 READLN - READLN inputs a list of variables from a file or the terminal keyboard. You should only use READLN on TEXT files. It differs from READ in that it reads an entire line of data up to a carriage return/line-feed pair. Given READLN(Data,Line), where Data is a file variable, READLN performs the following actions:

```
Line := Data^;
WHILE NOT EOLN(Data) DO
   GET(Data);
GET(Data)
```

In other words, until we reach the end of the line (a carriage return/line-feed pair), read data into the variable Line. The invocation takes the form:

```
READLN(file-identifier,list-of-variables);
```

where file-identifier is a file variable associated with the file you want to read from. If you omit the file-identifier:

```
READLN(list-of-variables);
```

READLN reads from the pre-declared file INPUT; that is, it reads from the terminal.

Separate READLN arguments with commas.

10.1.5.3 WRITE - The WRITE procedure writes a list of expressions to a file or a terminal display. To print a string, enclose it within single quotation marks. You must only use WRITE for TEXT files. WRITE does not write an end-of-line marker (carriage return/line-feed pair) after writing the specified data. To begin a new line, use the WRITELN procedure. The invocation takes the form:

```
WRITE(file-identifier,expression-list);
```

where file-identifier is a file variable associated with the AMOS file you want to write the data to, and expression-list is the data to be written. The expression list may contain string literals, constants, variables of type INTEGER, REAL, CHAR, PACKED ARRAY[1..n] OF CHAR, and STRING. For example:

```
WRITE(NewFile,'Two INTEGERs followed by STRING:',INT,12,'IsaString');
```

If you omit the file-identifier:

```
WRITE(expression-list);
```

WRITE assumes you want to write to the pre-declared file OUTPUT (the terminal display).

10.1.5.4 WRITELN - WRITELN outputs a list of expressions to a file or terminal. To print a string literal, enclose it within single quotation marks. You must only use WRITELN with TEXT files. WRITELN differs from WRITE in that it writes an end-of-line marker (carriage return/line-feed pair) after writing the specified data. The invocation takes the form:

```
WRITELN(file-identifier,expression-list);
```

where file-identifier is a file variable, and expression-list is a list of expressions to be written. Separate the WRITELN arguments with commas. If you omit the file-identifier:

```
WRITELN(expression-list);
```

WRITELN assumes that you want to write to OUTPUT (the terminal display). You may write just a carriage return/line-feed to a file or terminal by omitting the expression-list:

```
WRITELN(file-identifier);
```

or:

```
WRITELN;
```

10.1.5.5 Formatting Output - AlphaPascal uses certain conventions for outputting data. STRING data and data of type CHAR are displayed with no leading spaces. Numbers are written differently, depending on whether they are REAL or INTEGER.

AlphaPascal will always print REAL and INTEGER numbers in decimal notation if the number is less than 12 digits. (If the number is larger than 12 digits, the number will be printed in scientific notation.) If the fractional part of a REAL number is greater than 11 digits, that number will be printed in scientific notation.

INTEGER numbers are printed as a sequence of digits, possibly preceded by a minus sign. INTEGER numbers are not printed with a leading space. REAL numbers are printed with a leading space, unless the number is negative, in which case the minus sign takes up that space. REAL numbers are accurate to nearly 12 digits. They are always rounded to 11 digits before being displayed to avoid annoying outputs such as 4.9999... instead of 5.

Both WRITE and WRITELN allow you to include optional arguments that give additional formatting instructions to AlphaPascal. The form of these arguments is (for both WRITE and WRITELN):

```
WRITE(expression1 : X : Y,expression2 : X : Y,....);
```

where X specifies a minimum field width, and Y specifies the number of digits to write after the decimal point. X and Y must both be of type INTEGER, and may be constants or variables. If you are not printing a REAL number, you may not specify the Y argument.

The minimum field width specifies the minimum number of spaces in which the number is to be printed. For example, if you want AlphaPascal to print the number right-justified in a field of ten spaces, use the value 10 for X. This gives the minimum field in which to print the number; if the number is larger than the specified field (for example, it is 11 digits), AlphaPascal will not truncate the number, but will use the necessary number of spaces.

If the number is a REAL number, you may also specify Y, the number of digits to be printed to the right of the decimal point. (For example, for dollar values, you would probably want to specify 2.) AlphaPascal rounds the REAL number to the specified number of places; it does not truncate it.

Although you will probably find the optional formatting arguments to be of most use in printing numbers, you may also print data of type CHAR or STRING specifying a minimum field width. By combining formatting of numbers and strings, you can construct tables and charts in which titles and numbers are neatly lined up. See the output of the sample program below for a simple example.

Here are some sample outputs (the "␣" symbol indicates a blank):

```
WRITE(1, -1, 1.0, -1.0);

        1-1␣1-1

WRITE(0.0, 1.0, 100.010, 0.0012, 1E12, 1.1E12, -1.23E-12);

        ␣0␣1␣100.01␣.0012␣1E12␣1.1E12-1.23E-12

WRITE(0.0:6:2, 1.0:6:2, 100.010:6:2, 0.0012:6:2, -1.23E-12:6:2);

        ␣␣0.00␣␣1.00␣100.01␣␣␣.00␣-0.00
```

Below we give a sample program that demonstrates both formatted output and the use of files:

```
PROGRAM FormatOutput;

VAR           Report : FILE OF REAL;
        Year,Profit : REAL;
                  I : INTEGER;

BEGIN { FormatOutput }
  OPEN(Report,'YTD.DAT',OUTPUT);        { Put data in file. }
  FOR I := 1 TO 5 DO
    BEGIN { Loop }
      WRITE('Enter Year: ');READLN(Year);
      Report^:=Year;
      PUT(Report);
      WRITE('Enter Profit: ');READLN(Profit);
      Report^:=Profit;
      PUT(Report)
    END { Loop };

  RESET(Report);                  { Open file again-- for input }

  WRITELN('Year': 6 ,'Profit' : 18 );    { Print header }
  WRITELN('-----------------------------');
  WRITELN;

  WHILE NOT EOF(Report) DO        { Print contents until End of file}
    BEGIN { While-loop }
      Year := Report^;
      GET(Report);
      Profit := Report^;
      GET(Report);
      WRITELN(Year : 6,Profit : 20 : 2);   { Format output }
    END { While-loop };
END { FormatOutput }.
```

The program above prints a neat table of the form:

```
   Year              Profit
 ------------------------------

   1971             650000.56
   1973            1205600.34
   1975            1865030.89
   1977          100450677.34
   1979           82380000.90
```

10.1.6 PAGE

The Page procedure writes a form-feed to the specified file. The invocation takes this form:

```
PAGE(file-identifier)
```

where file-identifier is a file variable.

10.1.7 RESET

The RESET procedure "rewinds" your file to the beginning. In effect, it performs a CLOSE and then OPENs the file for input. The invocation takes the form:

```
RESET(file-identifier);
```

where file-identifier is a file variable that is associated with the file you want to reset. As does OPEN, RESET inputs the first file component into the buffer variable for you.

10.1.8 REWRITE

The REWRITE procedure opens a file for output. In effect, it performs a CLOSE followed by an ERASE; then it opens the file for output. The invocation takes the form:

```
REWRITE(file-identifier);
```

where file-identifier is a file variable that is associated with the file you want to rewrite. REWRITE has the ability to generate file names if no file specification is associated with the specified file-identifier. These file names begin with TEMPAA.TMP, and go on to TEMPAB.TMP, TEMPAC.TMP, ... TEMPZZ.TMP. For example, the program:

```
PROGRAM TestRewrite;

VAR   NewFile : FILE OF CHAR;

BEGIN { TestRewrite }
  REWRITE(NewFile)        { No file specification associated with
                            NewFile };
  PFILE(NewFile)          { Print filespec now associated with NewFile }
END { TestRewrite }.
```

prints:

```
TEMPAA.TMP
```

10.2 SPECIAL FUNCTIONS AND PROCEDURES FOR FILE I/O

Standard Pascal gives you several functions and procedures that allow you to read and write data from a file (e.g., GET, PUT, READ, etc.). We talked about these routines in the sections above. Although you will often use most of the functions and procedures discussed in those earlier sections to transfer data between the terminal and your programs, AlphaPascal also provides a number of additional functions and procedures that allow you to work with AMOS disk files.

Using the functions and procedures we discuss below, you can search for, define, open and close sequential and random AMOS files. The functions and procedures we discuss in the following sections are:

LOOKUP	Searches for specified file; returns Boolean value.
OPEN	Opens file in input, output, or random mode.
OPENI	Opens file in input mode.
OPENO	Opens file in output mode.
OPENR	Opens file in random mode.
CLOSE	Closes file associated with specified file-identifier.
FSPEC	Returns number of characters in filespec; associates filespec with file-identifier.
EXTENSION	Forces specified extension into file specification.
GETFILE	Reads information in file specification.
SETFILE	Places information into file specification.
CREATE	Allocates random file blocks
SEEK	Positions random file to specified file record.
ERASE	Erases specified file from disk.
FILESIZE	Returns number of disk blocks used by file.
JOBDEV	Returns device user is logged into.
JOBUSER	Returns account user is logged into.
PFILE	Prints name of file associated with specified channel
RAD50	Converts three-character string to RAD50 format.
RENAME	Renames specified file.

10.2.1 Information on AMOS Files

The AMOS file system recognizes two major types of files: random and sequential. Creating, opening, and performing I/O for the two types of files differs somewhat, so it is important to understand the differences between them.

Before we discuss AMOS disk files, we would like to mention again that the pre-declared file-identifiers INPUT, OUTPUT, and KEYBOARD have associated with them special AMOS file specifications: TTY:, TTY:, and KBD:.

TTY: specifies your terminal. (For example, if you give TTY: as the file specification to the compiler listing option, $L, the compiler sends the listing to your terminal display.) The KBD: specification is equivalent to the TTY: specification except that it prevents input from being echoed to

the terminal display if the terminal is in Charmode. (See Section 11.2.1, "Charmode," for information on charmode.)

NOTE: The normal end-of-line separator is a carriage return. Normally, the monitor appends a line-feed character onto the end of a carriage return. If you are in Charmode and are using the KBD: device, the monitor does not automatically append a line-feed onto the end of a carriage return. Therefore, if you are using KBD: in Charmode you should use GETs and PUTs to retrieve data, since READ has a one-character lookahead buffer which will cause it to wait on the line-feed when it encounters a carriage return.

10.2.1.1 Random Files - Random file blocks are allocated contiguously on the disk, and access to such a file is random; that is, by computing an offset, the system can access any one record in that file without accessing any other record. Random file blocks are 512 bytes. To create a random file, you will use the CREATE procedure.

One advantage in using a random file is that access to that file is very efficient; using the SEEK procedure, you may randomly position to any record in that file without stepping through prior records. In addition, a random file is the only file which you may read from and write to without closing and opening it again.

Do not use READ and WRITE to get data from a random file; use GETs and PUTs. You should be aware that the order in which you do GET and PUT procedures makes a difference. If you do a GET and then a PUT to update information in a random file, the last record retrieved via a GET will be updated; if you do a PUT, and then do a GET, you will get the record after the one you just updated. A series of GETs will retrieve successive records in a random file just as it will a sequential file. A series of PUTs will write to successive records.

The EOF function does not return TRUE after the end of a random file has been reached; instead, an error is generated. This error will also be generated if you SEEK a record beyond the end of the file and then attempt a GET or PUT.

10.2.1.2 Sequential Files - Sequential file blocks are allocated in a linked list on the surface of the disk, with one word at the front of each block containing the disk address of the next block in the file. Access to such a file is sequential, since the system has to read each block in order to find out where a specific block is on the disk. Sequential file blocks are 510 bytes. The EOF function returns TRUE after the end of a sequential file has been reached.

10.2.1.3 Logical Records - Within each disk block of a file, you can have one or more "logical records." The size of a logical record is determined by your programs. For example, if a grouping of data in your data file is CustomerName, 30 bytes; CustomerAddress, 50 bytes; and, SocialSecurity, 9 bytes, your file logical records might be 89 bytes. (For information on blocking logical records into disk blocks, see Section 12.2.3, "CREATE.")

A random file record may not be larger than 512 bytes, and maximum random file size is 65535 records. A sequential file logical record can cross block boundaries, and so may be larger than 512 bytes.

10.2.1.4 Opening and Setting Up Files - The usual sequence of events for opening and using a file goes this way:

1. Declare a file variable. For example:

```
VAR   DataFile : FILE OF CHAR;
```

   This variable establishes the file "channel"; the communication line over which your program will transfer data in and out of the file associated with that channel. In our discussions below, the term "file-identifier" refers to the file variable associated with the actual AMOS file.

2. Before you can use an AMOS file, you have to associate the specification of that file with the file-identifier you have declared, and you must open the file. This tells AMOS what AMOS file you will be accessing via the file-identifier.

   An AMOS file specification consists of:

```
Device Unit Filename Extension Project-number Programmer-number
```

   For example:

```
DSK0:CUSTID.DAT[100,3]
```

   where DSK is the Device, 0 is the Unit, CUSTID is the Filename, DAT is the Extension, 100 is the Project-number, and 3 is the Programmer-number. You can use several procedures to associate the file specification with the file-identifier (e.g., FSPEC, SETFILE, EXTENSION). You can then use OPENI, OPENO, OPENR, RESET, or REWRITE to open the file. Or, you can combine these two steps by using OPEN, which takes the form:

```
OPEN(file-identifier, filespec, mode);
```

   where file-identifier is a file variable; filespec is the file specification in string literal or variable form, and mode (INPUT, OUTPUT, or RANDOM) tells AlphaPascal whether the file is going to be used for input, output, or (in the case of random files), random

update. With INPUT and RANDOM modes, besides associating the file-identifier with a file specification, OPEN also inputs the first record of the file for you.

3. Once you have set a file up to start doing I/O, you can use GETs and PUTs or READs and WRITEs to transfer data between your program and the file.

4. The final stage in using an AMOS file is to close it, using the CLOSE procedure. Closing the file makes sure that the last record updated in the file gets written out to the file, and makes the file available for being opened again. (You can't open an open file.) It also makes the file-identifier available for association with a possibly different AMOS file. Files are automatically closed when you leave the procedure in which they were declared.

A simple case of opening and reading a file might look something like this:

```
PROGRAM TestFile;

VAR     CustID   : FILE OF STRING { Declare file-identifier };
        UserFile : STRING;

BEGIN { TestFile }
  WRITE('Please enter name of your data file: ');
  READLN(UserFile);

  OPEN(CustID,UserFile,INPUT)    { Open the file; get the data };
  WRITE(CustID^)                 { Display data in buffer };
  CLOSE(CustID)                  { Close the file }
END { TestFile }.
```

The small program above asks the user for a file specification and opens that file. The actual process of using the OPEN procedure inputs the first record of that file into the buffer variable automatically assigned to the file-identifier, CustID^.

10.2.2 CLOSE

You will use the CLOSE procedure to close a sequential file that is open for output. Closing the file ensures that the last record will get written to the file; it also enters the file into the disk directory.

You may not OPEN a file that is already open, so if you have been using a sequential file for output, and you want to use it for input, you must first close it and then re-open it for input. The invocation takes the form:

```
CLOSE(file-identifier);
```

where file-identifier is the file variable associated with the AMOS file you want to close. For example, given:

```
VAR   TaxRecs : FILE OF CHAR;
```

once we have opened and used the AMOS file associated with TaxRecs, we must close it:

```
CLOSE(TaxRecs);
```

As your program leaves each procedure or function, any files declared in those routines are automatically closed for you. However, using the CLOSE procedure ensures that if you are forced to do a hasty and untidy exit from your program (for example, if a system error occurs), the last record of the file will be written when the CLOSE procedure is executed. Closing a file also makes its file-identifier available for use with another file.

10.2.3 CREATE

All random files must be pre-allocated on the disk before you can use them. (That means that their maximum size must be established before you use them. You can copy random files to sequential files and vice versa, so if you are in doubt about the ultimate size of a file that you are building, it is a good idea to write the data to a sequential file first, then copy the file to a random file after you know how many records have to be copied.)

The CREATE procedure allocates a random file. The invocation takes the form:

```
CREATE(file-identifier,size);
```

where file-identifier is a file variable associated with the AMOS file you want to create, and size is a variable of type INTEGER that designates the number of disk blocks you want the file to contain.

NOTE: You must associate an AMOS file specification with the file-identifier before using CREATE. (You may use FSPEC, SETFILE, or OPEN (with the RANDOM mode) to do so.) For example:

```
PROGRAM RandomFile;

VAR     RanFile : FILE OF STRING;
        Counter : INTEGER;

BEGIN { RandomFile }
  Counter := FSPEC(RanFile,'NEWFIL','DAT');
  CREATE(RanFile,20)
END { RandomFile }.
```

The program above creates the 20-block random file NEWFIL.DAT. The FSPEC function assigns the filespec NEWFIL.DAT to the file variable FILE RanFile.

NOTE: CREATE causes an error if the file you are creating already exists or if there are not enough contiguous blocks available for it to be allocated

on the disk.

If you wish to create a random file capable of holding X records of type T, then the number of blocks it will require is:

```
1 + X DIV (512 DIV SIZEOF(T))
```

10.2.4 ERASE

The ERASE procedure erases a file from the disk. The invocation takes the form:

```
ERASE(file-identifier);
```

where file-identifier is the file variable associated with the AMOS file you want to erase.

ERASE does not return an error if the specified file is not there. This makes it very useful for ensuring that the creation of new files will be successfully carried out. For example, since CREATE (see above) and OPENO return an error if the file you want to create already exists, you can use ERASE before using OPENO or CREATE to make sure that the file you want to open does not already exist. If the file doesn't exist, ERASE can't erase it, but no error is generated and no harm is done. If the file does exist, ERASE erases it, and leaves the way clear for OPENO and CREATE.

We've rewritten the small program in Section 10.2.3, "CREATE," to include the ERASE procedure:

```
PROGRAM TestErase;

VAR     RanFile : FILE OF CHAR;
        Counter : INTEGER;

BEGIN { TestErase }
  Counter := FSPEC(RanFile,'NEWFIL','DAT');
  ERASE(RanFile);       { Make sure file doesn't already exist }
  CREATE(RanFile,20)
END { TestErase }.
```

10.2.5 EXTENSION

The EXTENSION procedure forces the specified extension in the specification of the AMOS file associated with the specified file variable. The invocation takes the form:

```
EXTENSION(file-identifier,ext);
```

where file-identifier is the file variable associated with the AMOS file, and ext is a string literal or variable that designates the extension you want to force to the file specification. For example:

```
PROGRAM TestExtension;

VAR     TheFile : FILE OF CHAR;
        Counter : INTEGER;

BEGIN { TestExtension }
  Counter := FSPEC(TheFile,'NEWFIL','DAT');
  EXTENSION(TheFile,'LST');
  PFILE(TheFile)
END { TestExtenstion }.
```

The program above associates the AMOS file NEWFIL.DAT with the file-identifier TheFile. Then it uses the EXTENSION procedure to change the extension from DAT to LST. (Notice the use of the PFILE procedure to print the AMOS file specification.) NOTE: EXTENSION does not change the extension of the file on the disk, it only changes the extension of the file specification associated with the file-identifier.

10.2.6 FILESIZE

The FILESIZE function returns the number of disk blocks taken up by the AMOS file associated with the specified file variable. You must have previously used the OPEN or LOOKUP procedure on the AMOS file. The invocation takes the form:

```
FILESIZE(file-identifier);
```

where file-identifier is a file variable. For example:

```
PROGRAM TestFileSize;

VAR     TheFile : FILE OF CHAR;
        Counter : INTEGER;

BEGIN { TestFileSize }
  Counter := FSPEC(TheFile,'BIGFIL','DAT');
  CREATE(TheFile,50);
  WRITELN('The number of disk blocks is: ',FILESIZE(TheFile))
END { TestFileSize }.
```

First the program above creates the random file BIGFIL.DAT, then it prints:

```
The number of disk blocks is: 50
```

10.2.7 FSPEC

The FSPEC function performs two main functions: it associates the specified file variable with the specified AMOS file, and it returns the number of characters in the specified variable or string literal that make up the file specification part. The invocation takes the form:

```
FSPEC(file-identifier, filename, default-extension);
```

where file-identifier is a file variable with which you want to associate the AMOS filespec, filename gives the name of the AMOS file, and default-extension gives the extension you want to use if no extension is supplied. For example:

```
PROGRAM TestFspec;

VAR     DataFile : FILE OF CHAR;
        UserFile : STRING;
        Counter : INTEGER;

BEGIN { TestFspec }
  WRITE('Please enter file specificaton: ');
  READLN(UserFile);
  Counter :=  FSPEC(DataFile,UserFile,'DAT');
  WRITELN('Number of characters: ',Counter);
  WRITE('File spec is: ');
  PFILE(DataFile)
END { TestFspec }.
```

You can use FSPEC to input an entire command line, not just a file specification. If the user of the program enters:

```
NEW,OLD
```

the program prints:

```
Number of characters: 3
File spec is: NEW.DAT
```

Then we can use the DELETE procedure:

```
DELETE(UserFile,1,Counter)
```

to leave the remainder of the user input ('OLD') in UserFile.

(Note that we used PFILE to print the name of the file associated with the file variable DataFile, and that the FSPEC function added the default extension of .DAT.)

10.2.8 GETFILE

The GETFILE procedure allows you to find out exactly what file specification is associated with a specific file-identifier. The invocation takes the form:

```
GETFILE(file-identifier, Dev, Unit, File1, File2, Ext, Proj, Prog);
```

The arguments are declared INTEGER variables. The data is returned as integers, because file specifications are stored internally by AMOS in a special numeric form called "RAD50." RAD50 format compresses three bytes of ASCII data into two bytes of numeric data. (In other words, GETFILE returns the file specification in RAD50 form.) File1 and File2 are the first three and second three RAD50 characters of the filename.

Although GETFILE may not sound too useful by itself, by doing GETFILEs on more than one file you can compare elements of the specifications for those files, and by using SETFILE (described in Section 10.2.20, below), you can actually change those elements. For example, consider the program below. It asks for the specifications of two data files needed for input; if those two files do not exist on the same device and unit, the program moves the files to the System Device, DSK0:, so that they <u>are</u> on the same disk device and unit.

```
PROGRAM;

TYPE  DataFile = FILE OF CHAR;

VAR   Dev, Unit, FileA, FileB, Ext, Proj, Prog : INTEGER;
        Dev1, Unit1, FileA1, FileB1, Ext1, Proj1, Prog1 : INTEGER;
        MailLabel, Addresses : DataFile

BEGIN { Main Program }
  WRITELN('Enter the specifications of your two data files');
  WRITELN;
  WRITE('File #1: ');READLN(UserSpec);
  WRITE('File #2: ');READLN(UserSpec1);

  OPEN(MailLabel,UserSpec,OUTPUT);      { Open the user-specified files }
  OPEN(Addresses,UserSpec1,OUTPUT);
  GETFILE(MailLabel,Dev,Unit,FileA,FileB,Ext,Proj,Prog);
  GETFILE(Addresses,Dev1,Unit1,FileA1,FileB1,Ext1,Proj1,Prog1);

  { See if files are on the same disk }
  IF (Dev <> Dev1) OR (Unit <> Unit1) THEN
    BEGIN
      WRITE('You have asked for files: '); PFILE(UserSpec);
      WRITE(' and '); PFILE(UserSpec1); WRITELN; WRITELN;

      WRITELN('Both of your data files must be on the same');
      WRITELN('device and unit; we are moving them both to DSK0:.');

      SETFILE(MailLabel,RAD50('DSK'),RAD50('0'),FileA,FileB,Ext,Proj,Prog);
      SETFILE(Addresses,RAD50('DSK'),RAD50('0'),FileA1,FileB1,
        Ext1,Proj1,Prog1)
    END;
  WRITE('Your files are: ');PFILE(UserSpec);
  WRITE(' and ');PFILE(UserSpec1);
END { Main Program }.
```

10.2.9 JOBDEV

The JOBDEV function takes two INTEGER variable arguments. The invocation takes the form:

```
JOBDEV(Dev,Unit);
```

JOBDEV returns in Dev the device you are currently logged into (in RAD50 form), and returns in Unit the device unit you are currently logged into (in INTEGER form).

10.2.10 JOBUSER

The JOBUSER function takes two INTEGER variable arguments. The invocation takes the form:

```
JOBUSER(Project,Programmer);
```

It returns in Project the project number (in decimal) you are logged into, and returns in Programmer the programmer number (in decimal) you are logged into.

10.2.11 LOOKUP

The LOOKUP function returns a TRUE or a FALSE depending on whether the specified file exists. The invocation takes the form:

```
LOOKUP(file-identifier);
```

where file-identifier is the file variable associated with the AMOS file you are looking for. Since several file procedures generate an error if the file specified to them already exists (e.g., OPENO, CREATE), while other procedures generate an error if the file doesn't exist, doing a LOOKUP before one of these procedures is a good idea. For example:

```
PROGRAM LookForIt;

VAR   FileID : FILE OF CHAR;
        Target : STRING;
        Query  : CHAR;
        X      : INTEGER;

BEGIN { LookForIt }
  WRITE('Enter the file you want to write to: '); READLN(Target);
  X := FSPEC(FileID,Target,'DAT');
  IF LOOKUP(FileID)
    THEN
      BEGIN
        WRITE('That file already exists. Destroy it? (Y or N): ');
        READLN(Query);
        IF Query = 'N' THEN EXIT(PROGRAM);
        ERASE(FileID);
        WRITELN('File erased.')
      END;
   OPENO(FileID);
   WRITELN('File ',Target,' opened for output.')
END { LookForIt }.
```

The program above checks to see if the file specified by the user already exists. If the file exists, the user is asked to decide whether or not to save the file, or get rid of it and start a new one of that name.

10.2.12 OPEN

The OPEN procedure opens a sequential file for input or output, or opens a random file for both input and output. The invocation takes the form:

```
OPEN (file-identifier, filespec, mode);
```

where file-identifier is a file variable, and filespec is a valid AMOS file specification. Mode may be INPUT, OUTPUT, or RANDOM, and specifies whether the file is to be a sequential file used for input or output, or (in the case of RANDOM), whether it is to be a random file used for input and output both. If you are using OPEN in OUTPUT mode, it deletes the specified file if it already exists. Default extension is .DAT. For example:

```
OPEN (INP, 'TEST', RANDOM);
```

associates the AMOS file TEST.DAT with the file-identifier INP, and opens the random file for input and output. Most of the sample programs in this chapter use the OPEN procedure.

NOTE: OPEN in INPUT or RANDOM mode inputs the first record into the buffer variable.

10.2.13 OPENI

OPENI is a variation of the OPEN procedure; it opens a sequential file for input. The invocation takes the form:

```
OPENI(file-identifier);
```

where file-identifier is a file variable associated with the AMOS file you want to open. If the file does not exist or if the file-identifier has not been associated with an AMOS file (via an FSPEC or SETFILE) OPENI generates an error. OPENI inputs the first record of the file into the buffer variable.

10.2.14 OPENO

OPENO is a variation of the OPEN procedure; it opens a sequential file for output. The invocation takes the form:

```
OPENO(file-identifier);
```

where file-identifier is the file variable associated with the AMOS file you want to open. If the file already exists or if the file-identifier has not been associated with an AMOS file (via FSPEC or SETFILE), OPENO generates an error.

10.2.15 OPENR

OPENR is a variation of the OPEN procedure; it opens a random file for input and output. The file must exist, and may not already be open. The invocation takes the form:

```
OPENR(file-identifier);
```

where file-identifier is the file variable associated with the AMOS file you want to open.

NOTE: OPENR inputs the first record of the file into the buffer variable.

10.2.16 PFILE

The PFILE procedure displays on your terminal the AMOS file specification associated with the specified file-identifier. The invocation takes the form:

```
PFILE(file-identifier);
```

where file-identifier is a file variable associated with the AMOS file whose specification you want to see. (Several of the sample programs in this chapter use PFILE.)

10.2.17 RAD50

The AMOS system stores much of the information used by the file system in a special form, called "RAD50." RAD50 compresses three bytes of ASCII data into two bytes of numeric data. The RAD50 procedure converts a string into RAD50 form. This is necessary if you are going to use the SETFILE procedure, since SETFILE expects several of its arguments in RAD50 form. For example, if you are planning to use SETFILE to change the filename of an AMOS file specification, you will do a GETFILE to get that specification:

```
GETFILE(TheFile,Dev,Unit,Filnam1,Filnam2,Ext,Proj,Prog);
```

The elements Dev, Filnam1, Filenam2, and Ext are returned in RAD50 form. Now, you will do a SETFILE to change the Filename:

```
SETFILE(TheFile,Dev,Unit,RAD50('NEW'),RAD50('NAM'),Ext,Proj,Prog);
```

leaving the rest of the elements as they were.

10.2.18 RENAME

The RENAME procedure allows you to rename an AMOS file. The invocation takes the form:

```
RENAME(file-identifier,newname);
```

where file-identifier is a file variable associated with the AMOS file you want to rename, and newname is a string variable or a string literal. For example, if the AMOS file CURRNT.DAT is associated with the file-identifier AccountsFile:

```
RENAME(AccountsFile,'BACKUP.LST');
```

renames the AMOS file CURRNT.DAT to BACKUP.LST. By varying the fields you supply to RENAME, you can rename just the extension, just the filename, or both. For example, if the AMOS file OLDDAT.DAT is associated with the file-identifier MailBox:

```
RENAME(MailBox,'.BAK');
```

renames OLDDAT.DAT to OLDDAT.BAK, and

```
RENAME(MailBox,'ARCHIV');
```

renames OLDDAT.DAT to ARCHIV.DAT.

10.2.19 SEEK

The SEEK procedure allows you to position a file pointer to a specific record in a random file for file I/O. The invocation takes the form:

```
SEEK(file-identifier,recordnum);
```

where file-identifier is a file variable associated with the random file we want to access, and recordnum is an integer variable or constant that specifies the number of the record to access. (The first record is record #0.)

REMEMBER: SEEK does not input a record into the buffer variable; it just positions the file pointer.

10.2.20 SETFILE

SETFILE takes the same arguments as GETFILE, but it puts information into the file specification. It also associates the specified file-identifier with the specified AMOS file. The invocation takes the form:

```
SETFILE(file-identifier, Dev, Unit, File1, File2, Ext, Proj, Prog);
```

For example:

```
SETFILE(NewFile,0,0,F1,F2,RAD50('LST'),0,0);
```

The sample above is changing the extension of the AMOS file associated with NewFile to .LST. NOTE: Specifying a zero for both the project AND the programmer number tells AMOS to use the current default project-programmer number (the account you are logged into). Specifying a zero for both the device AND the unit number forces AMOS to use the default device specification (the device and unit you are logged into). If you specify a device (e.g., RAD50('DSK')), you can tell AMOS to use the default unit, by specifying a negative 1 for the unit. For a more lengthy example of the use of SETFILE, see Section 10.2.8, "GETFILE."

10.3 SAMPLE PROGRAM TO DEMONSTRATE FILE HANDLING

The program below is an example of a programming solution to a very common business problem: the need for an efficient way of reading in, organizing, and maintaining employee information. Our sample program below uses random file techniques to maintain the following information for a user-defined number of employees: name, age, and sex. The employee records are maintained in alphabetical order by name of employee. You may add, delete, change, list, or display employee records.

10.3.1 Sample Run

A sample run looks like this (We will underline the information that the user of the program types in):

PRUN DEMO (RET)

< The screen clears >

AlphaPascal Random File Demonstration

Do you wish to (re-)create employee file? Y (RET)
How many records to you wish to use? 20 (RET)

< The screen clears >

Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: A (RET)
Last Name = ZUCKER (RET)
First Name = SUE ELLEN (RET)
Middle Initial = R (RET)
How old is SUE ELLEN? 23 (RET)
Is SUE ELLEN male? Y (RET)

Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: A (RET)
Last Name = ARROWSMITH (RET)
First Name = JACK (RET)
Middle Initial = C (RET)
How old is JACK? 51 (RET)
Is JACK male? Y (RET)

Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: A (RET)
Last Name = ALLEN (RET)
First Name = EDNA (RET)
Middle Initial = N (RET)
How old is EDNA? 35 (RET)
Is EDNA male? N (RET)

```
Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: L (RET)

ALLEN, EDNA N: 35 years old, sex: female
ARROWSMITH, JACK C: 51 years old, sex: male
ZUCKER, SUE ELLEN R: 23 years old, sex: male

Total of 3 employee(s)

Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: C (RET)
Last Name = ZUCKER (RET)
First Name = SUE ELLEN (RET)
Middle Initial = R (RET)
How old is SUE ELLEN? 23 (RET)
Is SUE ELLEN male? N (RET)

Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: L (RET)

ALLEN, EDNA N: 35 years old, sex: female
ARROWSMITH, JACK C: 51 years old, sex: male
ZUCKER, SUE ELLEN R: 23 years old, sex: female

Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: Q (RET)

                    < The screen clears >

Leaving AlphaPascal Random File Demonstration
```

10.3.2 The Program

```
PROGRAM EmployeeMaintenance;

TYPE
        NameRecord = RECORD
                       First: STRING[11];
                       Middle: CHAR;
                       Last: STRING[15];
                     END {NameRecord} ;

        EmpRecType = (Control,Data,Unused);
        EmpRecord = RECORD
                     CASE EmpRecType OF
                      Data: (
                        Name: NameRecord;
                        Age: INTEGER;
                        Sex: (Male, Female);
                        NextDataRecord: INTEGER);
                      Control: (
                        FirstDataRecord: ARRAY ['A'..'Z'] OF INTEGER;
                        FirstUnusedRecord: INTEGER);
                      Unused: (
                        NextUnusedRecord: INTEGER);
                    END {EmpRecord} ;

        EmpFileType = FILE OF EmpRecord;

{Global Variables}
VAR     EmpFile: EmpFileType;
        RecNum, PreviousRecNum: INTEGER;
        ControlRecord: EmpRecord;

FUNCTION SameNames(Name1,Name2: NameRecord): BOOLEAN;
{Returns TRUE if Name1 = Name2}
BEGIN
  SameNames :=      (Name1.First = Name2.First)
                AND (Name1.Middle = Name2.Middle)
                AND (Name1.Last = Name2.Last)
END {SameNames} ;
```

```
FUNCTION Find(Name: NameRecord): BOOLEAN;
{Searches for specified record in EmpFile.
 Returns true if found, leaving file positioned at desired record.}
BEGIN
        RecNum := ControlRecord.FirstDataRecord[Name.Last[1]];
        PreviousRecNum := 0;
        WHILE RecNum <> 0 DO
         BEGIN  SEEK(EmpFile,RecNum);
                GET(EmpFile);
                IF SameNames(Name,EmpFile^.Name)
                 THEN BEGIN Find:=TRUE; EXIT(Find) END;
                PreviousRecNum := RecNum;
                RecNum := EmpFile^.NextDataRecord;
         END;
        Find := FALSE;
END {Find} ;

FUNCTION Remove(Name: NameRecord): BOOLEAN;
{Deletes specified record in EmpFile.
 Returns false if not found.}
VAR     NextRecNum: INTEGER;
BEGIN
        Remove := TRUE;
        IF Find(Name) THEN
         BEGIN
          NextRecNum := EmpFile^.NextDataRecord;
          EmpFile^.NextUnusedRecord := ControlRecord.FirstUnusedRecord;
          ControlRecord.FirstUnusedRecord := RecNum;
          PUT(EmpFile);
          IF PreviousRecNum = 0
           THEN ControlRecord.FirstDataRecord[Name.Last[1]]
                 := NextRecNum
           ELSE
            BEGIN
                SEEK(EmpFile,PreviousRecNum);
                GET(EmpFile);
                EmpFile^.NextDataRecord := NextRecNum;
                PUT(EmpFile);
            END;
          SEEK(EmpFile,0);
          EmpFile^:=ControlRecord;
          PUT(EmpFile);
         END
        ELSE {Name not found} Remove := False;
END {Remove} ;
```

```
FUNCTION NamePrecedesName(Name1,Name2: NameRecord): BOOLEAN;
{Returns TRUE if Name1 <= Name2}
BEGIN
  NamePrecedesName :=
        IF Name1.Last <= Name2.Last
         THEN TRUE
         ELSE IF Name1.Last = Name2.Last
               THEN IF Name1.First <= Name2.First
                     THEN TRUE
                     ELSE IF Name1.First = Name2.First
                           THEN Name1.Middle <= Name2.Middle
                           ELSE FALSE
               ELSE FALSE;
END {NamePrecedesName} ;

FUNCTION Add(Employee: EmpRecord): BOOLEAN;
{Adds specified employee record to EmpFile.
 Returns false if no room remains to add record.}
VAR     InsertionPointFound: BOOLEAN; NewRecNum: INTEGER;
BEGIN
        Add := TRUE;
        RecNum := ControlRecord.FirstDataRecord[Employee.Name.Last[1]];
        PreviousRecNum := 0;
        InsertionPointFound := (RecNum = 0);
        WHILE NOT InsertionPointFound DO
         BEGIN  SEEK(EmpFile,RecNum);
                GET(EmpFile);
                IF NamePrecedesName(Employee.Name,EmpFile^.Name)
                 THEN InsertionPointFound := TRUE
                 ELSE BEGIN PreviousRecNum := RecNum;
                            RecNum := EmpFile^.NextDataRecord;
                            InsertionPointFound := (RecNum = 0);
                      END;
         END {Search for insertion point} ;
        IF RecNum <> 0 THEN
         IF SameNames(Employee.Name,EmpFile^.Name) THEN
          BEGIN Employee.NextDataRecord := EmpFile^.NextDataRecord;
                EmpFile^ := Employee;
                PUT(EmpFile);
                EXIT(Add);
          END;
        IF 0 = (NewRecNum := ControlRecord.FirstUnusedRecord) THEN
         BEGIN  Add := False {EmpFile is full};
                EXIT(Add);
         END;
        SEEK(EmpFile,NewRecNum);
        GET(EmpFile);
        ControlRecord.FirstUnusedRecord := EmpFile^.NextUnusedRecord;
        EmpFile^:=EmployeeRecord;
        EmpFile^.NextDataRecord := RecNum;
        PUT(EmpFile);
```

```
        IF PreviousRecNum = 0 THEN
         BEGIN  SEEK(EmpFile,0);
                ControlRecord.FirstDataRecord[Employee.Name.Last[1]]
                 := NewRecNum;
                EmpFile^ := ControlRecord;
                PUT(EmpFile);
         END
        ELSE
         BEGIN  SEEK(EmpFile,PreviousRecNum);
                GET(EmpFile);
                EmpFile^.NextDataRecord := NewRecNum;
                PUT(EmpFile);
         END;
END {Add} ;

PROCEDURE CreateEmployeeFile(Size: INTEGER);
{Create/Recreate Employee File with specified number of employee records}
VAR     X,SizeInBlocks: INTEGER; CH: CHAR;
BEGIN
        SizeInBlocks := 1 + (Size+1) DIV (512 DIV SIZEOF(EmpRecord));
        X := FSPEC(EmpFile,'EMPFIL','DAT');
        CLOSE(EmpFile); {Close file if it is open}
        ERASE(EmpFile); {Erase file if it already exists}
        CREATE(EmpFile,SizeInBlocks);
        OPENR(EmpFile);
        ControlRecord.FirstUnusedRecord := 1;
        FOR CH := 'A' TO 'Z' DO ControlRecord.FirstDataRecord[CH] := 0;
        EmpFile^ := ControlRecord;
        PUT(EmpFile);
        FOR X := 1 TO Size-1 DO
         BEGIN  EmpFile^.NextUnusedRecord := X+1;
                PUT(EmpFile);
         END;
        EmpFile^.NextUnusedRecord := 0;
        PUT(EmpFile);
        CLOSE(EmpFile);
END {CreateEmployeeFile} ;

PROCEDURE OpenEmpFile;
BEGIN   OPEN(EmpFile,'EMPFIL',RANDOM);
        ControlRecord := EmpFile^;
END;

FUNCTION Yes(Message: STRING): BOOLEAN;
VAR     Answer: STRING;
BEGIN
        WRITE(Message,' '); READLN(Answer); LCS(Answer);
        IF Answer = 'y' OR Answer = 'yes' THEN Yes := TRUE
        ELSE IF Answer = 'n' OR Answer = 'no' THEN Yes := FALSE
        ELSE Yes := Yes('?Please answer yes or no:');
END {Yes} ;
```

```
PROCEDURE Introduction;
VAR     Quantity: INTEGER;
BEGIN
        CRT(-1,0);      {Clear Screen}
        WRITELN('  AlphaPascal Random File Demonstration');
        WRITELN;
        WRITELN;
        IF Yes('Do you wish to (re-)create employee file?') THEN
         BEGIN  WRITE('How many records to you wish to use? ');
                READLN(Quantity);
                WHILE Quantity < 1 OR Quantity > 100 DO
                 BEGIN  WRITE('?Please enter a number between 1 and 100: ');
                        READLN(Quantity);
                 END;
                CreateEmployeeFile(Quantity);
         END;
        OpenEmpFile;
        CRT(-1,0);      {Clear screen}
END {Introduction} ;

PROCEDURE GetName(VAR Name: NameRecord);
{Note: UCS only works on strings, and Middle is of type CHAR}
VAR     S:STRING[1];
BEGIN
        WITH Name DO
         BEGIN  WRITE('Last Name = '); READLN(Last); UCS(Last);
                WRITE('First Name = '); READLN(First); UCS(First);
                WRITE('Middle Initial = '); READLN(S); UCS(S);
                Middle := IF S='' THEN ' ' ELSE S[1];
         END;
END;

PROCEDURE GetEmployeeInfo(VAR Employee: EmpRecord);
BEGIN
 WITH Employee DO
   BEGIN
        WRITE('How old is ',Name.First,'? ');
        READLN(Age);
        WRITE('Is ',Name.First);
        Sex :=  IF Yes(' male?')
                 THEN Male ELSE Female;
   END;
END {GetEmployeeInfo} ;
```

```
PROCEDURE ShowEmployeeInfo(Employee: EmpRecord);
BEGIN
  WITH Employee,Name DO
    BEGIN
        WRITE(Last,', ',First,' ',Middle,':  ');
        WRITE(Age,' years old, ');
        WRITELN('sex: ',CASE Sex OF
                          Male: 'male';
                          Female: 'female';
                        ELSE '');
    END;
END;

PROCEDURE ProcessRequests;
VAR     Option: CHAR;

        PROCEDURE ListEmployees;
        VAR     CH: CHAR; Count: INTEGER;
        BEGIN
                Count := 0;
                WRITELN;
                FOR CH := 'A' TO 'Z' DO
                 BEGIN  RecNum := ControlRecord.FirstDataRecord[CH];
                        WHILE RecNum <> 0 DO
                          BEGIN SEEK(EmpFile,RecNum);
                                GET(EmpFile);
                                ShowEmployeeInfo(EmpFile^);
                                RecNum := EmpFile^.NextDataRecord;
                                Count += 1;
                          END;
                 END;
                WRITELN; WRITELN('Total of ',Count,' employee(s)');
        END {ListEmployees} ;

        PROCEDURE AddEmployee;
        VAR     Employee: EmpRecord;
        BEGIN
                GetName(Employee.Name);
                IF Find(Employee.Name) THEN
                 BEGIN  WRITELN('?Employee already on file');
                        EXIT(AddEmployee);
                 END;
                GetEmployeeInfo(Employee);
                IF NOT Add(Employee) THEN WRITELN('?Not enough room to add');
        END {AddEmployee} ;
```

```
        PROCEDURE ChangeEmployee;
        VAR     Name: NameRecord;
        BEGIN
                GetName(Name);
                IF Find(Name) THEN
                 BEGIN  ShowEmployeeInfo(EmpFile^);
                        GetEmployeeInfo(EmpFile^);
                        PUT(EmpFile);
                 END
                ELSE WRITELN('?Not found');
        END {ChangeEmployee} ;

        PROCEDURE DeleteEmployee;
        VAR     Name: NameRecord;
        BEGIN
                GetName(Name);
                IF NOT Remove(Name) THEN WRITELN('?Not found');
        END {DeleteEmployee} ;

        PROCEDURE Inquire;
        VAR     Name: NameRecord;
        BEGIN
                GetName(Name);
                IF Find(Name) THEN ShowEmployeeInfo(EmpFile^)
                              ELSE WRITELN('?Not found');
        END {Inquire} ;

BEGIN {ProcessRequests}
        REPEAT
         WRITE(
         'Enter option [A)dd, C)hange, D)elete, I)nquire, L)ist, Q)uit]: ');
         READLN(Option);
         CASE Option OF
          'a','A': AddEmployee;
          'c','C': ChangeEmployee;
          'd','D': DeleteEmployee;
          'i','I': Inquire;
          'l','L': ListEmployees;
          'q','Q': EXIT(ProcessRequests);
          ELSE WRITELN('?Invalid option');
         WRITELN;
        UNTIL FALSE {i.e., until EXIT}
END {ProcessRequests} ;
```

```
PROCEDURE Termination;
BEGIN
        CRT(-1,0);      {Clear screen}
        WRITELN('Leaving AlphaPascal Random File Demonstration');
END {Termination} ;

BEGIN {Program}
        Introduction;
        ProcessRequests;
        Termination;
END {Program} .
```

10.3.3 Program Organization

We would just like to point out that the program above could have been broken up into modules and linked as separate files. In fact, it would have been a good idea to do so. If we were going to break it up, we might consider taking the first two global type declarations and putting them into include files (see below). (For information on include files, see Section 4.3.2.2, "The Include Option ($I).") Then we might have made the FIND function a module, FIND.PAS.

10.3.3.1 The AMOS file NAMREC.INC -

```
TYPE    NameRecord = RECORD
                        First: STRING[11];
                        Middle: CHAR;
                        Last: STRING[15];
                     END { NameRecord };
```

10.3.3.2 The AMOS file EMPREC.INC -

```
TYPE    EmpRecType = (Control, Data, Unused);
        EmpRecord = RECORD
                     CASE EmpRecType OF
                      Data: (
                        Name: NameRecord;
                        Age: INTEGER;
                        Sex: (Male, Female);
                        NextDataRecord: INTEGER);
                      Control: (
                        FirstDataRecord: ARRAY ['A'..'Z'] OF INTEGER;
                        FirstUnusedRecord: INTEGER);
                      Unused: (
                        NextUnusedRecord: INTEGER);
                    END {EmpRecord} ;

        EmpFileType = FILE OF EmpRecord;
```

10.3.3.3 The AMOS file FIND.PAS -

```
MODULE FIND;
{$I NAMREC.INC}
{$I EMPREC.INC}

EXTERNAL FUNCTION SameNames
        (Name1, Name2: NameRecord): BOOLEAN;

EXTERNAL VAR
        EmpFile : EmpFileType;
        RecNum, PreviousRecNum: INTEGER;

FUNCTION Find(Name: NameRecord): BOOLEAN;
{Searches for specified record in EmpFile.
 Returns true if found, leaving file positioned at desired record.}
BEGIN
        RecNum := ControlRecord.FirstDataRecord[Name.Last[1]];
        PreviousRecNum := 0;
        WHILE RecNum <> 0 DO
         BEGIN SEEK(EmpFile,RecNum);
               GET(EmpFile);
               IF SameNames(Name,EmpFile^.Name)
                THEN BEGIN Find:=TRUE; EXIT(Find) END;
               PreviousRecNum := RecNum;
               RecNum := EmpFile^.NextDataRecord;
         END;
        Find := FALSE;
END {Find} ;
.
```

CHAPTER 11

MISCELLANEOUS FUNCTIONS AND PROCEDURES

The functions and procedures described in this chapter perform a variety of functions such as allowing your programs to position the cursor on the terminal screen and manipulating dynamic variables. The functions and procedures discussed in this chapter are:

CHR	Convert ASCII value to its character representation
ORD	Returns ordinal number of element in scalar type
PRED	Returns predecessor (i.e., previous item) of scalar type
SUCC	Returns sucessor (i.e., next item) of scalar type
KILCMD	Abort command file execution
NEW	Creates new dynamic variable
MARK	Marks element on the heap
RELEASE	Releases element on the heap
CRT	Position screen cursor, and enable certain terminal display options
CHARMODE	Sets terminal into Charmode; suppresses echoing
LINEMODE	Returns terminal from Charmode to line mode
INCHARMODE	Returns Boolean value telling you whether you are in Charmode or not

11.1 BASIC FUNCTIONS AND PROCEDURES

11.1.1 CHR

All characters displayed by the computer are members of the ASCII character set, and have a number (called the ASCII value) associated with them. The CHR function returns the ASCII character associated with a specified ASCII value. It accepts a positive, decimal INTEGER argument and returns a CHAR result. The function invocation takes this form:

```
CHR(number);
```

For example:

```
WRITELN(CHR(65));
```

prints the character A. (65 is the decimal ASCII value of the ASCII character "A".)

11.1.2 KILCMD

It is often convenient to set up command files that automatically invoke a series of system commands and Pascal programs. (Remember that a command file is a text file; each line contains data or a valid AMOS file specification. To execute the entire set of command and program invocations contained in the command file, supply just the name of the command file at AMOS command level.)

The KILCMD procedure tells PRUN to abort any command file execution. You probably will use KILCMD if an error occurs that would make continuing the execution of the commmand file awkward. The invocation takes this form:

```
KILCMD;
```

As an example of the use of KILCMD, consider the command file PCL that accompanies this release of AlphaPascal. The PCL command file compiles and links a Pascal source file. Suppose you supply to PCL the name of a source file that does not exist. If the compiler can't compile your program, then PLINK can't link it. So, CMPILR itself contains a KILCMD procedure call that is executed if a compilation fails; the system stops any command file being executed and returns you to AMOS command level.

For information on error handling and writing your own errortrap routine, see Chapter 14, "Systems Functions and Procedures."

11.1.3 MARK

MARK is used in combination with RELEASE to store and release dynamic variables allocated via NEW (see below) in a stack-like structure called the "heap." The invocation of MARK takes this form:

```
MARK(variable-identifier);
```

where variable-identifier specifies a pointer variable that points to any type (typically, INTEGER). MARK returns the current state of the heap. That is, it returns the current address of the top of the heap.

A "heap" or "stack" can be considered as a sequential list in which items may only be inserted or deleted from one end of the list. Items are deleted in the reverse of the order in which they were entered on the stack.

The NEW procedure allocates dynamic variables on the heap. For example, if you use MARK, then perform a NEW, then use MARK again, MARK will return two different values, since the top of the heap changes when you allocate the dynamic variable.

By doing a MARK followed by a NEW, you have a value that tells you where on the heap the variable allocated by NEW is located. The way to free up heap-space used by the dynamic variables allocated via NEW is to use RELEASE (see Section 11.1.7, below).

NOTE: Be very careful when using MARK and RELEASE; unwise use of these procedures can leave you pointing to areas of memory that are not part of the heap, thus causing unpleasant and unpredictable results.

11.1.4 NEW

The NEW procedure allocates a dynamic variable. The invocation takes the form:

```
NEW(variable-identifier);
```

where variable-identifier is the pointer to the variable allocated by NEW. To access the variable allocated via NEW, use the pointer variable variable-identifier^. (For more information on NEW and dynamic variables, see Section 7.2.8, "Pointer Type.") The sections on MARK and RELEASE in this chapter give information on using MARK, NEW, and RELEASE to allocate and de-allocate dynamic variables on the "heap."

11.1.5 ORD

The ORD(X) function returns the ordinal number of the argument in the scalar data type of which X is a member. Accepts arguments of type CHAR or user-defined scalar types. Returns an INTEGER result. The function invocation takes this form:

```
ORD(variable-identifier or constants);
```

For example, each character displayed by the computer has a numeric value associated with it (called the ASCII value), which specifies its position in the set of ASCII characters. If you use the ORD function on an ASCII character, ORD will return to you the ASCII value of that character (that is, its ordinal number in the ASCII character set). For example:

```
WRITELN(ORD('A'));
```

(Changed 30 April 1981)

returns the decimal number 65, the ASCII value of the character 'A'. You may also include an identifier for a user-defined scalar type. For example:

```
PROGRAM TestOrd;

TYPE DAYSOFTHEWEEK = (MON,TUE,WED,THUR,FRI);

BEGIN { TestOrd }
  WRITELN('Ordinal number of THUR is: ',ORD(THUR));
  WRITELN('Ordinal number of D is: ',ORD('D'))
END { TestOrd }.
```

The program above prints the ordinal number of the character "D" in the ASCII character set, and the ordinal number of "THUR" in the user-defined scalar type DAYSOFTHEWEEK. (NOTE: The ordinal numbers for the elements of DAYSOFTHEWEEK are: MON = 0, TUE = 1, WED = 2, THUR = 3, FRI = 4.)

11.1.6 PRED

The PRED function returns the predecessor of the specified scalar argument. The invocation of the PRED function takes this form:

```
PRED(element);
```

For example, let's say that we defined the scalar type Cardinal to contain the elements: First, Second, and Third:

```
TYPE  Cardinal = (First, Second, Third);
```

Since the elements of a scalar data type are ordered, we can find out what element is previous to the specified item by using the PRED function. For example:

```
IF PRED(Second) = First THEN WRITELN('Correct!');
```

The value returned by PRED is not a variable or an expression; therefore, trying to use WRITE or WRITELN to display that value causes an error. (That is, you may not say: WRITELN(PRED(Second)).)

```
PROGRAM TestPred;

TYPE  Daysoftheweek = (Mon,Tue,Wed,Thu,Fri);
VAR   Day : Daysoftheweek;

BEGIN { TestPred }
  Day := Tue;
  IF PRED(Day) = Mon THEN WRITELN('Today is Tuesday');
  Day := PRED(Day);
  IF Day = Mon THEN WRITELN('It''s Blue Monday!')
END { TestPred }.
```

When you run the program above, it prints:

```
Today is Tuesday
It's Blue Monday!
```

11.1.7 RELEASE

The RELEASE procedure is used with MARK and NEW to use dynamic variables with a stack-like structure called the "heap." (See Section 11.1.3, "MARK," for information on the heap.) It de-allocates the dynamic variable at the specified heap location. The RELEASE invocation takes the form:

```
RELEASE(variable-identifier);
```

where variable-identifier is the same argument as that supplied to MARK. For example, if you use MARK to get the current state of the heap, use NEW to allocate a dynamic variable (which advances the top of the heap past the value returned by the previous MARK), and then use RELEASE with the value returned by the previous MARK, RELEASE de-allocates the dynamic variable from the heap. A picture might help to clarify:

```
Procedure                   The Heap

 NEW(V0)                       V0
 MARK(LocationV1)          ------
 NEW(V1)                       V1
 MARK(LocationV2)          ------
 NEW(V2)                       V2
```

Then:

```
Use RELEASE(LocationV2)
Use RELEASE(LocationV1)
```

RELEASE(LocationV2) de-allocates V2; RELEASE(LocationV1) de-allocates V1. V0 is left on the stack in the example above. You cannot RELEASE a dynamic variable in the middle of the heap; you may only release variables from the bottom of the list.

NOTE: Be very careful when using MARK and RELEASE; unwise use of these procedures can leave you pointing to areas of memory that are not part of the heap, which can cause severe problems.

11.1.8 SUCC

The SUCC procedure allows you to determine the successor element to the sepcified scalar constant. The invocation takes the form:

```
SUCC(element);
```

where element is a variable-identifier or constant of a scalar type. For example:

```
PROGRAM;

VAR  Int : INTEGER;
     Dat : (YES, NO, Y, N);

BEGIN
   WRITE('Enter integer: '); READLN(Int);
   WRITELN(SUCC(Int));
   Dat := YES;
   IF SUCC(Dat) = NO THEN WRITELN('YES')
END.
```

If you enter the number 11 to the program above, it prints:

```
12
YES
```

(See also Section 11.6, "PRED," for more information on manipulating scalar types.)

11.2 SPECIAL TERMINAL DISPLAY PROCEDURES

11.2.1 CHARMODE

The CHARMODE procedure allows you to set the terminal of the user of your program into character mode. When a terminal is in character mode, your program is able to read keyboard input a character at a time, even before a terminating carriage return is typed. (Assembly language programmers on the AMOS system may recognize this input mode as "image mode.") The invocation of this procedure takes this form:

```
CHARMODE;
```

Character mode is useful for checking special input such as passwords, since the characters are not echoed at the time they are input, but when read (via a GET or READ). To inhibit echoing, use the pre-declared KEYBOARD file identifier.

NOTE: Character editing (such as RUBs or Control-Us) doesn't work when the terminal is in character mode. To return a terminal to the normal mode, use the LINEMODE procedure (discussed in Section 11.2.3, below). When your program exits to monitor level, AMOS automatically puts the terminal back into LINEMODE.

11.2.2 CRT

The CRT function allows you to position the cursor on the terminal screen. In addition, you can also select certain terminal-handling options (such as clear screen, delete character, etc.).

The function invocation takes this form:

```
CRT(Arg1,Arg2);
```

where Arg1 and Arg2 are integers. If Arg1 is positive, the CRT function assumes that you want to position the cursor on the screen; if Arg2 is negative, CRT assumes that you want to use one of the extended screen-handling options.

11.2.2.1 Cursor Positioning - If the first argument you supply to CRT is positive, then the CRT function reads both arguments as the X,Y row/column coordinates specifying the screen position where you want the cursor positioned. (The top left-hand corner of the screen is specified by the coordinates 1,1.) For example, the function:

```
CRT(12,35);
```

positions the cursor at the 12th row and 35th column of the screen.

NOTE: If you supply row and column coordinates that are out of range for your terminal, unpredictable results could occur.

11.2.2.2 Extended Screen Display Options - If the first argument supplied to CRT is negative, the CRT function assumes that you want to use the extended terminal-handling options specified by the second argument. For example, the function:

```
CRT(-1,0);
```

tells CRT to select option #0, the clear-screen option.

The screen-handling options provided are:

Code	Function
0	Clear screen
1	Cursor home (upper left corner)
2	Cursor return (column 0 without line-feed)
3	Cursor up one row
4	Cursor down one row
5	Cursor left one column
6	Cursor right one column
7	Lock keyboard
8	Unlock keyboard
9	Erase to end of line
10	Erase to end of screen
11	Protect field (reduced intensity)
12	Unprotect field (normal intensity)
13	Enable protected fields
14	Disable protected fields
15	Delete line
16	Insert line
17	Delete character
18	Insert character
19	Read cursor address
20	Read character at current cursor address
21	Start blinking field
22	End blinking field
23	Start line drawing mode (enable alternate character set)
24	End line drawing mode (disable alternate character set)
25	Set horizontal position
26	Set vertical position
27	Set terminal attributes

NOTE: You should be aware that these options can be selected only if your particular terminal and terminal driver program are capable of carrying them out. (For example, not all terminals can perform an erase-to-end-of-screen function.) Note that most terminals do not support all of the options listed above; unsupported options will be ignored by your terminal driver.

11.2.3 INCHARMODE

The INCHARMODE function returns a Boolean result. If it returns a TRUE, then you are in charmode; a FALSE indicates that you are in linemode. (See the paragraph below.)

11.2.4 LINEMODE

The LINEMODE procedure returns a terminal to the normal input mode after it has been set into character mode via the CHARMODE procedure (discussed in Section 11.2.1, above). The invocation takes this form:

```
LINEMODE;
```

While in line mode, all input is ended by a carriage return, and character editing is enabled. Character echoing occurs as you type the characters, not when they are read.

CHAPTER 12

MATHEMATICAL FUNCTIONS

The following functions accept one or more numeric arguments. For information on invoking functions and on writing your own functions, see Section 6.6, "Function and Procedure Declarations."

12.1 TRIGONOMETRIC FUNCTIONS

12.1.1 COS(X)

Cosine trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result. Argument must be in radians.

12.1.2 SIN(X)

Sine trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result.

12.1.3 TAN(X)

Tangent trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result.

12.1.4 ARCCOS(X)

Arc cosine trigonometric function. Computes the inverse cosine function. (See COS above.) Accepts a REAL or INTEGER argument and returns a REAL result. X must be greater than or equal to -1, and less than or equal to 1.

12.1.5 ARCSIN(X)

Arc sine function. Computes the inverse sine function. (See SIN above.) Accepts a REAL or INTEGER argument and returns a REAL result. X must be greater than or equal to -1, and less than or equal to 1.

12.1.6 ARCTAN(X)

Arc tangent trigonometric function. Computes the inverse tangent function. (See TAN above.) Accepts a REAL or INTEGER argument and returns a REAL result.

12.2 HYPERBOLIC TRIGONOMETRIC FUNCTIONS

12.2.1 COSH(X)

Hyperbolic cosine trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result. Argument must be in radians.

12.2.2 SINH(X)

Hyperbolic sine trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result.

12.2.3 TANH(X)

Hyperbolic tangent trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result.

12.2.4 ARCCOSH(X)

Hyperbolic arc cosine trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result. (See ARCCOS above.) X must be greater than or equal to 1.

12.2.5 ARCSINH(X)

Hyperbolic arc sine trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result. (See ARCSIN above.)

12.2.6 ARCTANH(X)

Hyperbolic arc tangent trigonometric function. Accepts a REAL or INTEGER argument and returns a REAL result. (See ARCTAN above.) The absolute value of X must be less than 1.

12.3 MISCELLANEOUS MATHEMATICAL FUNCTIONS

12.3.1 ABS(X)

Computes the absolute value of the argument. Accepts one INTEGER or REAL argument, and returns an INTEGER or REAL result. For example:

```
WRITELN(ABS(-32.123));
```

displays the answer:

```
32.123
```

12.3.2 EXP(X)

Exponential function. Computes e to the X power, where e is the base of natural logarithms. Accepts a REAL or INTEGER argument; returns REAL result.

12.3.3 EXPONENT(X)

Computes K such that X = J * 2^K, where J is greater than or equal to .5, and less than 1. Accepts a REAL argument.

12.3.4 FACTORIAL(X)

Computes the factorial of X. Accepts a REAL argument; returns a REAL result. For example:

```
FACTORIAL(6.0)
```

returns 720. (720 = 6*5*4*3*2*1.)

12.3.5 LN(X)

Computes the natural (Napierian) logarithm. Accepts a REAL or INTEGER argument; returns a REAL result. Computes logarithm to the base e. (e = 2.71828...)

12.3.6 LOG(X)

Computes the log base ten of the argument. Accepts a REAL or INTEGER argument; returns a REAL result.

12.3.7 ODD(X)

Tests for odd value. Accepts INTEGER argument; returns a BOOLEAN result. If X is odd, ODD returns TRUE; if X is even, ODD returns FALSE.

12.3.8 POWER(X,Y)

Computes X to the Y power. Accepts two REAL numbers; returns a REAL result. For example:

```
POWER(2.0,3.0)
```

returns 8. You can also use POWER to compute the Nth root of a number--POWER(X,1.0/N).

For example, to find the cube root (third root) of 256.12:

```
POWER(256.12,1.0/3.0)
```

(Changed 30 April 1981)

12.3.9 PWROFTEN(X)

Returns the value of ten raised to the power of X. Accepts an INTEGER or REAL value; returns a REAL value. Accepts fractions and negative numbers. For example:

```
PWROFTEN(3)
```

returns 10 to the third, or 1000.

12.3.10 PWROFTWO(X)

Returns the value of two raised to the power of X. Accepts an INTEGER value and returns a REAL value. Number must be greater than zero. For example:

```
PWROFTWO(3)
```

returns 2 to the third power, or 8.

12.3.11 RANDOMIZE

Randomizes the starting seed of the RND function (see below). It takes no arguments. For example:

```
RANDOMIZE;
```

12.3.12 RND

Returns a random REAL number between 0 and 1. It takes no arguments. For example:

```
PROGRAM TestRND;
{ Generate 20 random integers between 1 and 10 }
VAR   I : INTEGER;
BEGIN
    RANDOMIZE;
    WRITELN('Random numbers between 1 and 10:');
    FOR I := 1 TO 20 DO
       BEGIN
          WRITELN(TRUNC((RND*10)+1))
       END
END.
```

12.3.13 ROUND(X)

Rounds-off X. Accepts a REAL argument; returns an INTEGER result. For example, ROUND(23.78) returns 24; ROUND(23.45) returns 23.

12.3.14 SHIFT(X,Y)

Performs binary multiplication by shifting left the binary representation of the number specified by the first argument the number of places specified by the second argument. For example:

```
SHIFT(7,2);
```

returns the answer 28. (The binary number 111 (7 decimal) shifted left two places is the binary number 11100 (28 decimal).) X and Y must be of type INTEGER.

12.3.15 SQR(X)

Computes the square of X. For example, SQR(8) returns 64. Accepts REAL or INTEGER argument and returns an INTEGER or REAL result.

12.3.16 SQRT(X)

Computes non-negative square root of argument. Argument may be INTEGER or REAL; result is REAL. X must be greater than or equal to zero. Accepts a REAL or INTEGER argument; returns a REAL result.

12.3.17 STR(X) and STR(X,a,b)

Converts numeric values to STRING. Accepts a REAL or INTEGER number, and returns a STRING.

You may optionally supply STR with two INTEGER arguments that tell STR how to format a converted number:

```
STR(Number,X,Y);
```

or:

```
STR(Number,X);
```

where X specifies a minimum field width and Y specifies the number of digits to write after the decimal point. If the number is larger than the field specified by X, PASCAL does not truncate the number, but prints the number using the necessary number of digit positions.

(Changed 30 April 1981)

(If "Number" is INTEGER, you may not specify Y.) These two variations of STR perform formatting in exactly the same way as WRITE and WRITELN, except that they do not generate a leading space for positive numbers. For example, given the REAL data 123.44:

```
WRITELN(STR(123.44,10,4));
```

returns the string:

```
  123.4400
```

where the number is right-justified in a field of ten blanks, and four digits are written to the right of the decimal point.

12.3.18 TRUNC(X)

Truncates X. Accepts REAL argument; returns INTEGER result. (For example, TRUNC(24.3) returns the integer 24.)

12.4 SAMPLE PROGRAM TO PAD A NUMBER WITH LEADING ZEROS

Below is a useful procedure to pad a number with leading zeros along with a sample program that makes use of it:

```
PROGRAM Format;
VAR   S : STRING;
        { The procedure call Format(String,Left,Right,Number) formats
          the number with Left zero-filled digits before the decimal
          point and Right zero-filled digits after the decimal point.
          A trailing space or minus sign is added to indicate the sign
          of the number.  Illegal arguments generate an error to
          ERRORTRAP. }
PROCEDURE Format(VAR X : STRING; Left,Right : INTEGER; Num : REAL);
   VAR   Pow : REAL;
   BEGIN { Procedure Format }
      IF Left > 11 OR Left <= 0 THEN ERROR(1);
      Pow := PWROFTEN(Left);
      IF ABS(Num) >= Pow THEN ERROR(1) { Value range error };
      X := STR(Pow + ABS(Num),0,Right);
         { Force leading zeros by adding power of ten and converting
           to STRING. }
      DELETE(X,1,1);   { Remove leading 1 }
      X := IF Num < 0 THEN CONCAT(X,'-') ELSE CONCAT(X,' ')
   END { Format };
BEGIN { Main program }
  Format(S,5,2,-12.7); { Return answer in S }
  WRITELN('Format(5,2,-12.7) = ',S);
  WRITELN('Result should be 00012.70-')
END { Main program }.
```

(Changed 30 April 1981)

CHAPTER 13

STRING AND CHARACTER ARRAY FUNCTIONS AND PROCEDURES

This chapter contains descriptions of the standard functions and procedures you can use on data that have been declared as type STRING or packed arrays of type CHAR. These functions and procedures have been pre-declared for you by AlphaPascal. For a full list of all functions and procedures, refer to Appendix A, "Quick Reference to AlphaPascal."

These are the functions and procedures described in this chapter:

For data of type STRING:

CONCAT	Concatenates specified strings
COPY	Copies specified string (or partial string) into another string
DELETE	Deletes specified number of characters from string
INSERT	Inserts specified string (or partial string) into another string
LENGTH	Returns number of characters in string
LCS	Converts upper case string to lower case
POS	Returns position of specified character in string
STRIP	Removes trailing spaces from string
UCS	Converts lower case string to upper case
VAL	Converts a string to a REAL number.

For data of type PACKED ARRAY OF CHAR:

FILLCHAR	Fills specified string with specified character
MOVELEFT	Copies specified number of characters beginning with left of array over to specified array
MOVERIGHT	Copies specified number of characters beginning with right of array over to specified array
SCAN	Returns position of specified character in array

13.1 STRING INTRINSICS

Below are the functions and procedures that you can use on data of type STRING.

13.1.1 CONCAT

The CONCAT function returns a string that contains the contents of all of the specified string(s). The function invocation takes this form:

```
CONCAT(String1,String2,...,StringN);
```

where you may specify one or more strings to be concatenated.

For example:

```
PROGRAM TestConcat;

VAR   Destination, Source1, Source2, Source3 : STRING;

BEGIN { TestConcat }
   Source1 := '"Nevermore!"';
   Source2 := 'the Raven, ';
   Source3 := 'Quoth ';
   Destination := CONCAT(Source3,Source2,Source1);
   WRITELN(Destination)
END { TestConcat }.
```

The program above prints:

```
Quoth the Raven, "Nevermore!"
```

13.1.2 COPY

The COPY function creates a new string of the specified number of characters whose contents are taken from the specified source string, starting at the specified index. The function invocation takes this form:

```
COPY (Source-string,Index,Size-of-returned-string);
```

For example:

```
PROGRAM TestCopy;

VAR     Source, Target : STRING;
              Position : INTEGER;

BEGIN {TestCopy}
   Source := 'Jonathan R. Smith';
   Position := POS('S',Source) { Find position of last name };
   Target := COPY(Source,Position,5);
   WRITELN('The customer last name is: ',Target);
   WRITELN('Last-name position in source string is: ',Position);
END {End TestCopy}.
```

The program above prints:

```
The customer last name is: Smith
```

and:

```
Last-name position in source string is: 13
```

(Notice that we used the POS function, discussed below in Section 13.1.7, to determine the position in the source string of the character 'S'.)

13.1.3 DELETE

The DELETE procedure removes the specified number of characters from the source string, starting at the specified position. The procedure invocation takes this form:

```
DELETE (Source-string, Index, Number-of-characters);
```

where Source-string must be a string variable.

For example:

```
PROGRAM TestDelete;

VAR             Source : STRING;
        Position, Size : INTEGER;

BEGIN { TestDelete }
   Source := 'Now is the time for all good men!';
   Position := POS('all',Source);
   DELETE (Source, Position + 3, 9);
   WRITELN(Source)
END { TestDelete }.
```

The program above prints the string:

```
Now is the time for all!
```

13.1.4 INSERT

The INSERT procedure inserts the specified string into a specified destination string. It begins the insertion at the specified position in the destination string. The invocation of this procedure takes the form:

```
INSERT(Insert-string,Destination-string,Index);
```

where Destination-string must be a string variable. For example:

```
PROGRAM TestInsert;

VAR   Insertion,Destination : STRING;

BEGIN { TestInsert }
   Destination := 'Customer name is: .';
   Insertion := 'Robert Allen';
   INSERT(Insertion,Destination,19);
   WRITELN(Destination)
END { TestInsert }.
```

The program above prints:

```
Customer name is: Robert Allen.
```

13.1.5 LCS

The LCS procedure converts upper case characters to lower case. The procedure invocation takes this form:

```
LCS(SourceString);
```

where SourceString is the string to be converted. For example:

```
PROGRAM TestLCS;

VAR   CustomerID : STRING[22];

BEGIN { TestLCS }
  CustomerID := 'Alfred J. Prufrock Jr.';
  LCS(CustomerID);
  WRITELN('Converted name is: ',CustomerID)
END { TestLCS }.
```

The program above prints:

```
Converted name is: alfred j. prufrock jr.
```

13.1.6 LENGTH

The LENGTH function returns the number of characters in the specified string. The function invocation takes this form:

```
LENGTH(Source-string);
```

For example:

```
PROGRAM TestLength;

VAR   State : STRING;

BEGIN { TestLength }
   State := 'California';
   WRITELN('Number of characters in state: ',LENGTH(State));
   WRITELN('Number of characters in zipcode: ',LENGTH('90247'));
END { TestLength }.
```

The program above prints:

```
Number of characters in state: 10
```

and:

```
Number of characters in zipcode: 5
```

13.1.7 POS

The POS function returns the position of the first occurrence of the specified characters in the specified source string. If POS can't find the specified characters, it returns a zero. The invocation of this function takes the form:

```
POS(Pattern,Source-string);
```

For example:

```
PROGRAM TestPos;

VAR   Source : STRING;

BEGIN { TestPos }
   Source := 'The requested account number is #AA234-567-23228';
   WRITELN('The account number begins at character position # ',
           POS('#AA',Source))
END { TestPos }.
```

The program above prints the message:

```
The account number begins at character position # 33
```

13.1.8 STRIP

The STRIP procedure strips trailing blanks from the specified string. (That is, STRIP removes any blanks that are at the end of the string.) The invocation takes the form:

```
STRIP(SourceString);
```

where SourceString must be a string variable. For example:

```
PROGRAM TestStrip;

VAR   Source : STRING;

BEGIN { TestStrip }
  Source := '25 characters          ';
  WRITELN('Before stripping:[',Source,']');
  STRIP(Source);
  WRITELN('After  stripping:[',Source,']')
END { TestStrip }.
```

The program above prints:

```
Before stripping:[25 characters          ]
After  stripping:[25 characters]
```

13.1.9 UCS

The UCS procedure converts lower case characters in a specified string to upper case. The procedure invocation takes the form:

```
UCS(SourceString);
```

where SourceString must be a string variable. For example:

```
PROGRAM TestUCS;

VAR   Title : STRING[30];

BEGIN { TestUCS }
  Title := 'fAmOus comPUters i HAve knOWn.';
  UCS(Title);
  WRITELN('Converted title is: ',Title)
END { TestUCS }.
```

The program above prints:

```
Converted title is: FAMOUS COMPUTERS I HAVE KNOWN.
```

13.1.10 VAL

The VAL procedure converts a string to a REAL number. The invocation takes the form:

```
VAL(SourceString);
```

where SourceString is a string variable that you want to convert to a number. For example:

```
PROGRAM TestVAL;

VAR     Price : STRING;
        Total : REAL;

BEGIN { TestVAL }
   WRITE('Enter price of object: ');
   READLN(Price);
   IF POS('.',Price) = 0 THEN WRITELN('The price is in whole dollars.');
   Total := VAL(Price);
   WRITELN('With 6% tax, the price is:',Total * 1.06:8:2)
END { TestVAL }.
```

The program above uses a string function, POS, on the string Price; it then converts Price to a REAL number (Total) so that it can perform a numeric operation on the value. (Notice the use of the optional parameters (":8:2") in the last WRITELN invocation to format the numeric answer in a field eight characters wide with two digits to the right of the decimal point.)

A sample run of the program looks like this:

```
Enter price of object: 560
The price is in whole dollars.
With 6% tax, the price is:  593.60
```

13.2 CHARACTER ARRAY INSTRINSICS

The procedures and strings listed below are for use on packed arrays of type CHAR. You must make sure that any string literal you assign to the array is the correct number of characters. For example, asssigning a string literal to an array of 24 elements will cause an error if that string literal has less than or more than 24 characters.

13.2.1 FILLCHAR

The FILLCHAR procedure modifies a character array by filling it with the specified character. The invocation for the procedure takes this form:

```
FILLCHAR (Destination,Length,Fill-character);
```

where Destination must be a variable. For example:

```
PROGRAM TestFillChar;

VAR      Destination : PACKED ARRAY [1..10] OF CHAR;
              Length : INTEGER;
           Character : CHAR;
                   I : INTEGER;

BEGIN { TestFillChar }
  Length := 10;
  Character := 'A';
  FILLCHAR(Destination,Length,Character);
  FOR I := 1 TO 10 DO
    WRITE(Destination[I])
END { TestFillChar }.
```

The program above fills the character array Destination with ten A's.

13.2.2 MOVELEFT and MOVERIGHT

The MOVELEFT and MOVERIGHT procedures move blocks of bytes in memory. They can be dangerous if not used correctly. (For example, if you tell one of these procedures that you want to move 20 bytes, but the destination array only contains 10 bytes, where do the extra 10 bytes go? Somewhere in memory?)

You will probably use MOVELEFT and MOVERIGHT most often to shift characters within a single array. You can also use them to move characters from one array of type CHAR to another.

MOVELEFT starts at the left of the specified source array, and moves bytes to the specified position in the destination array (also beginning at the left). MOVERIGHT moves bytes beginning with the right of the source array,

and beginning with the right of the specified locations in the destination array. You specify the source array, the destination array, and the number of bytes to move. (In the case of an array of type CHAR, one byte is one character.) By including subscripts, you may specify the locations in the source and destination arrays at which to start.

Of course, MOVELEFT and MOVERIGHT do not physically "move" the bytes; instead, they make a copy of the specified bytes from the source array into the specified locations of the destination array. The invocations of MOVELEFT and MOVERIGHT take this form:

```
MOVELEFT(Source-array,Destination-array,Number-of-bytes);
```

and:

```
MOVERIGHT(Source-array,Destination-array,Number-of-bytes);
```

where Destination must be a variable.

Given the same arrays and same subscripts, the results of MOVERIGHT and MOVELEFT will look exactly the same. For example, if Source is the packed array of CHAR "1234567890", and Destination is the packed array of CHAR "**********",

```
MOVELEFT(Source[6],Destination[6],5);

MOVERIGHT(Source[6],Destination[6],5);
```

will produce the same packed array: *****67890. The MOVELEFT procedure above moves the characters in this order: 6, 7, 8, 9, and 0. The MOVERIGHT procedure above moves the characters in this order: 0, 9, 8, 7, and 6. The only time this will become important is when you are moving characters within the same array.

For example:

```
PROGRAM TestMove;

VAR   Source : PACKED ARRAY [1..23] OF CHAR;

BEGIN { TestMove }
  Source := 'Days are never too long';
  MOVELEFT(Source[6],Source[1],10);
  WRITELN(Source);
  Source := 'Days are never too long';
  MOVERIGHT(Source[6],Source[1],10);
  WRITELN(Source)
END { TestMove }.
```

The program above prints:

```
are never ever too long
```

and:

```
ever ever ever too long
```

MOVERIGHT and MOVELEFT can produce radically different results, depending on the data you give them. You must be careful to choose the correct MOVE function for your particular application.

13.2.3 SCAN

The SCAN function returns the number of characters in the array from the beginning of the array until the specified character. (If the specified characters are not found, SCAN returns the number of characters in the entire array.) The function invocation takes this form:

```
SCAN (Length,Partial-expression,Source-array);
```

where Length gives the length of the array, Source-array specified the packed array of type CHAR that is to be searched, and Partial-expression takes the form:

```
<> character-expression
```

or:

```
= character-expression
```

For example:

```
PROGRAM TestScan;

VAR   Source : PACKED ARRAY [1..25] OF CHAR;

BEGIN { TestScan }
   Source := 'Error:30240 type RETURN  ';
   WRITELN('Error code starts after char #: ',SCAN(25,=':',Source))
END { TestScan }.
```

If the searched for character-expression is the first character of the array, SCAN returns a zero.

By specifying a negative length, you can tell SCAN to scan the array backward, from right to left. If the specified character appears in the array, SCAN then returns a negative number specifying the number of characters scanned from the right of the array before the specified character was reached. If you supply a negative length, be sure to also specify the position in the array at which you wish the search to start. For example:

```
WRITELN('It starts after character #',SCAN(-25,=':',Source[25]));
```

PART III

ADVANCED PROGRAMMING ON THE ALPHA PASCAL SYSTEM

CHAPTER 14

SYSTEMS FUNCTIONS AND PROCEDURES

The following functions and procedures will be of special use to the experienced AlphaPascal programmer. They allow you to determine the location and size of data objects in memory, to determine the amount of free memory left, and to handle system and file errors.

Other functions and procedures allow you to access system functions such as accessing the line printer spooler, mounting disks, reading the system clock, and reading, setting, and releasing multi-user file locks.

14.1 LOCATION

The LOCATION function returns an integer that corresponds to the absolute memory address of the specified variable. The invocation takes the form:

```
LOCATION(variable-identifier);
```

where variable-identifier specifies the variable whose memory address you wish to know. LOCATION accepts a variable of any type as an argument. LOCATION may not be used on packed fields.

14.2 SIZEOF

The SIZEOF function returns the size (in decimal bytes) of the specified item. The invocation takes this form:

```
SIZEOF(variable-or-type-identifier);
```

For example:

```
PROGRAM TestSizeOf;

TYPE  SampleRecord = RECORD
                       character: CHAR;
                       next: ^SampleRecord;
                     END;

BEGIN { TestSizeOf }
    WRITELN('Size of SampleRecord (in bytes) is:',SIZEOF(SampleRecord))
END { TestSizeOf }.
```

The program above prints:

```
Size of SampleRecord (in bytes) is: 4
```

14.3 MEMAVAIL

MEMAVAIL returns an integer corresponding to 3/4 the number of unused words remaining in the user partition. This number can be used to estimate how many items can be allocated by NEW before memory capacity is exceeded. You can use SIZEOF to determine how many bytes any particular object will require.

14.4 MAINPROG

MAINPROG is a boolean function that returns no arguments. It returns TRUE if the .PCF file is being used as a program, or FALSE if it is being used as a library.

MAINPROG can be used for debugging purposes. It can be used to write a program which can also be used as the library of a checkout program that makes sure that the functions and procedures defined in the original program (now a library to the checkout program) are implemented correctly. To do this, the program would have the form:

```
PROGRAM;

  ...declarations...

BEGIN
    ...initialization...

  IF MAINPROG THEN
      BEGIN
       {statements}
      END;
END.
```

The statements are only executed if the program is being used as a program and not as a library.

The checkout program testing the functions and procedures in the above program would then declare them as EXTERNAL functions and procedures in order to call them with test arguments.

14.5 SPOOL

SPOOL is an assembly language routine that you can call from Pascal to spool a disk file to the line printer(s). (To "spool" a file is to insert it into the printer queue, after which you can continue to do other things while your file waits in the queue for its turn to be printed.) SPOOL allows you to specify on which printer the file is to be printed, the number of copies to be printed, the form on which it is to be printed, whether the file is to be deleted after it is printed, etc.

The current version of SPOOL (AMOS version 4.4/AlphaPascal version 2.0, and later) is fully compatible with the current BASIC SPOOL subroutine. In other words, the only information you must supply to SPOOL is the specification of the file you want to print; all other parameters are optional. However, any unspecified arguments must be replaced by a null value (STRING null or INTEGER 0, based on the type of the argument). This is because Pascal functions and procedures require a fixed number of arguments.

The following definitions of switches and error codes are provided in the include file SPOOL.INC. To use, insert {$I SPOOL} into the appropriate place in your program.

14.5.1 Switches

To make life easier, switch values are available as constants. For a description of SWITCHES, see below. The constants would be:

```
  BANNER = 1;
NOBANNER = 2;
  DELETE = 4;
NODELETE = 8;
  HEADER = 16;
NOHEADER = 32;
      FF = 64;
    NOFF = 128;
    WAIT = 256;
```

14.5.2 Error Codes

The error codes returned by SPOOL are provided in a TYPE declaration at the beginning of a program. The TYPE command has the form:

```
TYPE SPOOLERROR = (SPOOLED, NOSPOOLERALLOCATED, BADPRINTERNAME,
                   OUTOFQUEUEBLOCKS, FILENOTFOUND);
```

14.5.3 Function Definition

Finally, the external function linkage definition is made as follows:

```
EXTERNAL FUNCTION SPOOL(F,P: STRING;
                        SW,CP: INTEGER;
                        FRM: STRING;
                        L,W: INTEGER): SPOOLERROR;
```

14.5.4 The SPOOL Subroutine

Call SPOOL via:

```
SPOOLCODE:=SPOOL(FILENAME,PRINTER,SWITCHES,COPIES,FORMS,LPP,WIDTH);
```

where:

SPOOLCODE — A variable of type SPOOLERROR which gets the completion code shown in the above TYPE declaration. If SPOOLCODE is not set to SPOOLED after the call is made, then an error of some kind occurred and the file was not printed.

FILENAME — A string variable or expression that gives the specification of the file you want to print.

PRINTER — A string variable or expression that gives the name of the printer to which you want to send the file. If PRINTER is a null string, SPOOL uses the default printer.

SWITCHES — An integer variable or expression that specifies various control switches and flags that affect the printing of the file. The control switches that SPOOL uses are exactly the same as the switches used by the monitor PRINT command. (See the "AMOS System Commands Reference Sheets" in the User's Information section of the AM-100 documentation packet for information on PRINT.)

Each switch you can use has a numeric code associated with it (see below). For example, the BANNER switch code is 1; the DELETE switch code is 4. Set control switches by putting the sum of the appropriate switch codes into the SWITCHES variable.

For example, if you want to use the BANNER and DELETE switches (to tell the line printer spooler program to print a banner page and delete the file after printing it), load SWITCHES with 5 (BANNER code + DELETE code). If you set SWITCHES to 0 (or do not specify one value of a switch pair), SPOOL uses the default switches for the selected printer.

Switch codes:

BANNER	1
NOBANNER	2
DELETE	4
NODELETE	8
HEADER	16
NOHEADER	32
FF	64
NOFF	128
WAIT	256

COPIES — An integer variable or expression that specifies the number of copies to be printed. If COPIES is 0, the line printer spooler program prints one copy.

FORMS — A string variable or expression that specifies the form on which the file is to be printed. If FORMS is a null string, the line printer spooler uses the NORMAL form.

LPP — An integer variable or expression that specifies the number of lines per page. SPOOL only uses this value if you have specified the HEADER switch in the SWITCHES variable. If you omit LPP, the spooler program uses the default value for the specified printer.

WIDTH — An integer variable or expression that specifies the width (in characters) of the print line (for header printing). If WIDTH is 0, the spooler program uses the default value for the specified printer.

14.6 XLOCK AND GETLOCKS

XLOCK and GETLOCKS are two assembly language subroutines that allow setting, clearing, and listing of multi-user file locks, like the equivalent BASIC subroutine XLOCK. In fact, the locks set by the Pascal XLOCK are the same as those set by the BASIC XLOCK. This means that Pascal and BASIC programs can be used to lock each other out.

For a lengthier discussion of the concept of "file locks," refer to FLOCK - BASIC Subroutine to Coordinate Multi-user File Access, (DWM-00100-16), and XLOCK - BASIC Subroutine for Multi-User Locks, (DWM-00100-11), in the BASIC Programmer's Information section of the AM-100 documentation packet. Briefly, however, a file lock is merely a convenience that allows a program

to check to see if a current file is in use. The reason for file locks is that multiple users will destroy the contents of a file if they access it at the same time. A file lock helps programs keep track of whether or not a file is currently being accessed. The program accessing the file "sets" a lock on the file to let other programs know that they must wait. Then, when the program leaves the file, it "clears" the lock, making the file accessible to other programs. It is important to stress that a file lock is not a security device. Multiple programs can access the same file whether or not file locks are set on that file; however, by checking file locks, a program can prevent a file from being damaged by refraining from accessing it while another program is using it.

It is not necessary to load the Pascal XLOCK routine into memory. However, the routine does require one word of data in system memory to link to the system queue list, which contains the locks set by XLOCK

This link is contained in the file DSK0:XLOCK.SYS[1,4]. This file should be loaded into system memory so that XLOCK may work.

If XLOCK.SYS is not loaded into system memory, then the AlphaBASIC subroutine XLOCK.SBR must be. This is because the AlphaBasic XLOCK will contain within itself a link to the system queue blocks which contain the locks set by both XLOCK routines. For more information, see the XLOCK documentation.

There is no problem if both XLOCK.SBR and XLOCK.SYS are in system memory at the same time. The Pascal XLOCK will use XLOCK.SBR, so that BASIC and Pascal are using the same list of locks.

The lock values defined below (LOCK1 and LOCK2) are required for each lock. LOCK1 is called the "MAJOR LOCK" and LOCK2 is called the "MINOR LOCK." If a value of 0 is set in either lock, then that lock becomes a wildcard and matches all values in that position. For instance, a LOCK1 of 3 and a LOCK2 of 0 locks out all locks with a LOCK1 of 3. Any other user that tries to use a LOCK1 of 3 will be locked out. If LOCK1 is set to 0, then all locks will be set. For more information, see a description of the BASIC XLOCK subroutine.

14.6.1 The XLOCK Subroutine

XLOCK is an external function. Therefore, it must be specified as such in the Pascal program that uses it:

```
EXTERNAL FUNCTION XLOCK(MD: XLOCKMODE;
                        LOCK1,LOCK2: INTEGER):INTEGER;
```

The type XLOCKMODE is defined as follows:

```
TYPE  XLOCKMODE = (SETLOCK, SETLOCKWAIT, CLEARLOCK)
```

(These two declarations can be included in a program by using {$I XLOCK}.) The result of the XLOCK call will be returned in the integer variable RETCODE:

```
RETCODE:= XLOCK(MODE,LOCK1,LOCK2);
```

Where MODE is one of the modes specified in XLOCKMODE, and LOCK1 and LOCK2 are integers containing the locks to be set.

If RETCODE is ever set to -1, this means that a bad mode was passed to XLOCK. This can happen if there was an error in setting up XLOCKMODE.

14.6.2 Setting a Lock

A lock is set using XLOCK mode SETLOCK. For instance, if the user had opened a file on channel 3 and was updating record 47, he might enter the following code into his Pascal file:

```
LOCK1:=3; {locking file 3}
LOCK2:=47; {locking record 47}
RETCODE:=XLOCK(SETLOCK,LOCK1,LOCK2);
```

If the lock was successful, then RETCODE is set to 0. If not, the job number of the job that has that lock is returned in RETCODE.

14.6.3 Setting a Lock (and Waiting Until it is Available)

It is sometimes necessary to wait for a lock to become clear. To do this, mode SETLOCKWAIT is used instead of SETLOCK. This mode, assuming the above example, is used as follows:

```
RETCODE:=XLOCK(SETLOCKWAIT,LOCK1,LOCK2);
```

If the lock is held by another user, the program will be put to sleep until it becomes available. When XLOCK returns to the user program, RETCODE will contain a 0 if the lock was allocated, or the user's job number if the lock already was allocated to him.

14.6.4 Clearing a Lock

After a lock is no longer needed (i.e. in the above example, moving to another record) it must be cleared so that other users have access to that lock. To clear a lock, the CLEARLOCK mode is used. Again, using the above example:

```
RETCODE:=XLOCK(CLEARLOCK,LOCK1,LOCK2)
```

RETCODE will contain the number of locks that were cleared. If RETCODE = 0 after the call, then no locks were cleared. If RETCODE = 1, then one lock was cleared. If RETCODE > 1, then a wildcard was specified in LOCK1 or LOCK2 and mucho locks were cleared.

14.6.5 The GETLOCKS Subroutine

GETLOCKS is an external procedure. Therefore, it must be specified as such in the Pascal program that uses it:

```
EXTERNAL PROCEDURE GETLOCKS(VAR LOCKQTY,JOBNUM: INTEGER;
                            VAR LOCKARRAY: LARRAY);
```

Type LARRAY is an array of type LOCK. LOCK is set up as follows:

```
TYPE  LOCK = RECORD JOB,LOCK1,LOCK2 : INTEGER END;
```

If X is a variable of type LOCK, then X.JOB is the job number that holds the lock. X.LOCK1 and X.LOCK2 are the lock valued of the lock. LARRAY is defined as follows:

```
TYPE  LARRAY = ARRAY[1..25] OF LOCK;
```

The variable LOCKARRAY may then be allocated for GETLOCKS to return the list of locks in:

```
VAR LOCKARRAY : LARRAY;
```

Be sure to set up type LARRAY as an array large enough to hold the maximum number of possible locks on your system. Since there is no range checking in external procedures or functions, LARRAY must be large enough to receive the maximum number of anticipated locks. Therefore, it is a good idea to set LARRAY to the number of queue blocks allocated in your system.

If there is a possibility that more than 25 locks may be set at a time when GETLOCKS (see below) is called, then it is necessary that the size of LARRAY be increased. The file DSK0:XLOCK.INC[7,5], which contains the definition of LARRAY, may be modified.

To get a list of locks, enter into your program:

```
GETLOCKS(LOCKQTY,JOBNUM,LOCKARRAY);
```

Where LOCKQTY is an integer that receives the number of set locks, JOBNUM is an integer that receives your job number, and LOCKARRAY is the array that receives the list of locks.

One thing you might do with this list of locks is list it. To do this:

```
FOR LOCKLIST:=1 TO LOCKQTY DO
   WITH LOCKARRAY[LOCKLIST] DO
      BEGIN {LIST LOCKS}
        WRITELN('JOB =   ',JOB);
        WRITELN('LOCK1 = ',LOCK1);
        WRITELN('LOCK2 = ',LOCK2);
      END; {LIST LOCKS}
```

14.7 XMOUNT

XMOUNT is an assembly language routine that allows you to mount a disk from within a Pascal program without leaving Pascal. You should call it whenever you change a disk and your Pascal program is going to use that disk. You must always mount a disk after you have changed it and before you write to it. Otherwise, the system will think that the old disk is still in the drive, and use the old disk's bitmap to find unused disk blocks.

14.7.1 Error Codes

The error codes returned by XMOUNT are specified in a TYPE declaration at the beginning of a program, having provided the form:

```
TYPE MOUNTERROR = (MOUNTED, UNMOUNTED, DEVNOTFOUND, BADHASH, NOVOLID);
```

Next, some variables will have to be defined. XMOUNT requires a string variable to contain the device specification and another string variable that will contain the volume ID of the newly mounted disk. XMOUNT will return an error code in a variable that should be of type MOUNTERROR:

```
VAR DEV,VOLID: STRING[10];
      RETCODE: MOUNTERROR;
```

Next, the function (XMOUNT) must be defined as follows:

```
EXTERNAL FUNCTION XMOUNT(D: STRING;
                   VAR V: STRING): MOUNTERROR;
```

{$I XMOUNT} will include the required TYPE and EXTERNAL FUNCTION definitions required by XMOUNT.

XMOUNT is then called via:

```
RETCODE:=XMOUNT(DEV,VOLID);
```

Where DEV is a string containing the device to be mounted (e.g., DEV:='DSK3:') and VOLID is the string variable used to receive the volume ID, if any.

14.7.2 Unmounting a Disk

A disk may be unmounted by specifying '/U' after the DEV spec (i.e. RETCODE:=XMOUNT('DSK23:/U',VOLID);). If a disk is to be unmounted, the '/U' must contain an upper case 'U'. When you unmount a disk, you prevent BASIC and most system programs from being able to access that device. Note that VOLID is included, even though it is not needed because a volume id is not returned when a disk is unmounted. VOLID is required at all times.

14.7.3 Error Codes

The error (or return) codes specified above have the following meanings:

14.7.3.1 MOUNTED - The device was successfully mounted and the volume ID is in VOLID (or whatever the second string was called).

14.7.3.2 UNMOUNTED - The device was successfully unmounted. VOLID is unchanged.

14.7.3.3 DEVNOTFOUND - The specified device was not defined at system generation, and is not in the system device table. VOLID is unchanged

14.7.3.4 BADHASH - The device was mounted, but it was a storage module device with a BADBLK.SYS. When the new BADBLK.SYS was read, it was found to contain a bad hash total. VOLID is unchanged.

14.7.3.5 NOVOLID - The disk was successfully mounted, but there was no volume ID on the disk. Note that MOUNTED and NOVOLID specify successful mounting of the disk. UNMOUNT specifies a successful UNMOUNT. DEVNOTFOUND and BADHASH indicate errors occurred while attempting to mount the disk. If either of these errors occur, you should not try to access that device!

14.8 TIME

The TIME procedure places the contents of the system clock into the two specified variables. (The system clock contents increment every sixtieth of a second on most systems, and every fiftieth of a second on other systems; the actual amount is specified by CLKFRQ in SYSTEM.INI.) Word1 contains the most significant part of the returned value. Word1 and Word2 must be declared INTEGER variables. The procedure invocation takes this form:

```
TIME(Word1,Word2);
```

For example:

```
PROGRAM TestTime;

VAR   First,Second : INTEGER;

BEGIN { TestTime }
   TIME(First,Second);
   WRITELN('The time is: ',First,',',Second)
END { TestTime }.
```

When the program above was run at 5:30:02 PM, it printed:

```
The time is: 56,2086
```

NOTE: Because the clock contents are stored as a 32-bit unsigned value, Word2 may sometimes be interpreted and displayed by the computer as a negative number.

14.9 TOD

TOD takes no arguments, and returns a real number corresponding to the number of seconds since midnight, according to the time of day. Internally, the time of day is converted from a two word integer to a real number, and then divided by the clock frequency defined in SYSTEM.INI. Therefore, the resolution on 60 cycle systems is to within .01666... seconds, and on 50 cycle systems is to within .02 seconds.

You will probably find TOD to be of most use for timing purposes or for calculating the time of day.

14.10 ERROR HANDLING PROCEDURES AND VARIABLES

Whenever an error occurs, the AlphaPascal system prints an appropriate message (including the location of the error) and aborts to AMOS. Whenever a user types a Control-C while his program is executing, execution is suspended and the user is allowed to choose among a series of options such as resuming his program, exiting to AMOS, or displaying a backtrace of suspended function and procedure invocations.

However, it is not always desirable to let the system perform error handling for you. You may wish to allow the user to type a Control-C in order to exit from some mode of a program, or in order to obtain a status report on the progress of your program in processing some task. It may be that you have an applications package in which the users of your package are unfamilar with AMOS... if an error occurs you may simply wish to print a message and return to the top level of your applications package. Or it may be that you enforce security on your system, and have an unattended program that you wish to LOGOFF automatically if an error occurs.

For these reasons and more, it is desirable for you to be able to write your own routine to handle a Control-C and error conditions. AlphaPascal allows you to do so, and the remainder of this section will attempt to provide you with the necessary information to write such a routine.

14.10.1 Including ERT.INC

In order to write your own error routine, you must include a special set of definitions with {$I ERT}. Doing so includes the following text:

```
TYPE
  INFOREC = RECORD
              XEQERR: INTEGER;
              FILERC: INTEGER;
              ERRFIB: ^TEXT;
              ... {Additional information for internal use only}
            END;

EXTERNAL PROCEDURE XERRORTRAP(VAR INFO: INFOREC);
EXTERNAL PROCEDURE STDERRORTRAP;
```

14.10.2 ERRORTRAP

To catch errors, you must write a global procedure of no arguments with the name ERRORTRAP. Here is a very simple example of such a procedure:

```
PROCEDURE ERRORTRAP;
BEGIN
        STDERRORTRAP;
END;
```

In this example, we simply call the standard system error handler STDERRORTRAP. In order to determine the nature of the error which has invoked errortrap, you must use the function ERRORINFO in conjunction with a variable declared as type ^INFOREC:

```
PROCEDURE ERRORTRAP;
   VAR   INFOP: ^INFOREC;
   BEGIN
           INFOP := ERRORINFO;
           WRITELN('?Error ',INFOP^.XEQERR');
           STDERRORTRAP;
   END;
```

In this example, we also display the error code corresponding to the error which occurred before calling the standard error handler. The list of error codes is as follows:

Error Code	Meaning
1	Value range error
3	Exit from uncalled procedure
4	Memory capacity exceeded
5	Integer overflow
6	Divide by zero
7	Bad pointer reference
8	(Control-c)
10	(I/O error)
11	Unimplemented runtime instruction
12	Floating point error
13	String overflow
14	Programmed HALT
15	Programmed breakpoint
16	ARCSIN(x) or ARCCOS(x) where abs(x) > 1
17	LOG(x) or LN(x) where x <= 0
18	SQRT(x) where x < 0
19	TAN(PI/2 + k*PI) is undefined for integer k (bad TAN argument)
20	ARCCOSH(X) where x < 1
21	FACTORIAL(x) where x is a negative integer
22	ARCTANH(x) where abs(x) >= 1
23	POWER(x,y) where x < 0 and y is a fraction

In the case of I/O errors, there is some additional information, namely the type of error and the file involved, which is available:

```
PROCEDURE ERRORTRAP;
    VAR   INFOP: ^INFOREC;  BADFILEP: ^TEXT;
    BEGIN
      INFOP := ERRORINFO;
      IF INFOP^.XEQERR = 10 {I/O Error} THEN
        BEGIN BADFILEP := INFOP^.ERRFIB;
              WRITE('?I/O error ',INFOP^.FILERC,' has occurred in ');
              PFILE(BADFILEP^);
              WRITELN;
        END;
      STDERRORTRAP;
    END;
```

In the above example, if an I/O error has occurred, we display the file error code (INFOP^.FILERC) and the name of the file involved (INFOP^.ERRFIB^). ERRFIB is a pointer to the most recently processed file, which is why we first save it in BADFILEP before writing to the terminal, otherwise our message would read

```
?I/O error xxx has occurred in TTY:
```

regardless of the actual file in which the error occurred.

Here is a list of the I/O error codes. They are the standard codes used by AMOS:

I/O Error Code	Meaning
1	File specification error
2	Insufficient free memory for INIT
3	File not found
4	File already exists
5	Device not ready
6	Device full
7	Device error
8	Device in use
9	Illegal user code
10	Protection violation
11	Write protected
12	File type mismatch
13	Device does not exist
14	Illegal block number
15	Buffer not INITed
16	File not open
17	File already open
18	Bitmap kaput
19	Device not mounted
20	Invalid filename

In the examples so far, we have always been calling STDERRORTRAP to handle our errors. STDERRORTRAP always aborts to AMOS without returning with the

exception of a Control-C followed by a command to resume. Thus the ERRORTRAP procedures themselves have aborted to AMOS in most circumstances.

In addition, STDERRORTRAP resets INFOP^.XEQERR to zero before returning if execution is to be resumed. This is because errors MUST NOT occur in the error handler itself for obvious reasons. AlphaPascal assumes it is executing an error handler whenever XEQERR is nonzero. If an error does occur within an error handler, the message

?Attempt to call ERRORTRAP while in ERRORTRAP

is displayed, a direct abort to AMOS is made without closing any open files. Thus, by reseting XEQERR to zero, STDERRORTRAP signals to AlphaPascal that error handling is finished and further errors are again acceptable.

Should you decide not to call STDERRORTRAP at all, please keep in mind the following points:

1. The only errors from which you may safely resume execution are 8 (a Control-C) and 10 (I/O error). An attempt to resume execution by returning from ERRORTRAP with any other errors will probably crash the system.

2. It is acceptable to use EXIT to abort some function or procedure, or your program, when any error occurs. Of course you can only EXIT to leave a function or procedure which is currently active, so you will probably want to have around some BOOLEAN variables to keep track of whether or not you are currently within routines which you might wish to EXIT from ERRORTRAP.

3. Remember to set XEQERR back to zero before leaving ERRORTRAP, otherwise your next error will abort to AMOS without calling ERRORTRAP.

14.10.3 XERRORTRAP

When STDERRORTRAP is called by entering a Control-C, it is possible to request a backtrace of suspended functions and procedures. This backtrace begins with the caller of the caller of STDERRORTRAP, which is usually the caller of ERRORTRAP, and hence the routine which was suspended. Thus, should STDERRORTRAP be called by a function or procedure local to your ERRORTRAP procedure, the backtrace will begin in the wrong place. This can be corrected by using XERRORTRAP, which takes a copy of the system INFOREC as its argument. It is used as follows:

```
PROCEDURE ERRORTRAP;
VAR   INFOP: ^INFOREC; INFO: INFOREC;
  PROCEDURE P1;
     BEGIN
        XERRORTRAP(INFO);
     END;
BEGIN
  INFOP := ERRORINFO;
  INFO := INFOP^;
  P1;
  INFOP^.XEQERR := INFO.XEQERR;
END;
```

Using XERRORTRAP, the backtrace will be displayed beginning with the caller of the routine which invokes ERRORINFO, thus producing a correct backtrace, even when called from an inner procedure. Begin at the caller of the procedure which set X to ERRORINFO.

14.10.4 ERROR

The procedure ERROR(x) takes an INTEGER x as argument and generates the corresponding system error. See the previous section for the list of error codes.

exception of a Control-C followed by a command to resume. Thus the ERRORTRAP procedures themselves have aborted to AMOS in most circumstances.

In addition, STDERRORTRAP resets INFOP .XEQERR to zero before returning if execution is to be resumed. This is because errors MUST NOT occur in the error handler itself for obvious reasons. AlphaPascal assumes it is executing an error handler whenever XEQERR is nonzero. If an occur does occur within an error handler, the message

?Attempt to call ERRORTRAP while in ERRORTRAP

is displayed, a direct abort to AMOS is made without closing any open files. Thus, by reseting XEQERR to zero, STDERRORTRAP signals to AlphaPascal that error handling is finished and further errors are again acceptable.

Should you decide not to call STDERRORTRAP at all, please keep in mind the following points:

1. The only errors from which you may safely resume execution are 8 (a Control-C) and 10 (I/O error). An attempt to resume execution by returning from ERRORTRAP with any other errors will probably crash the system.

2. It is acceptable to use EXIT to abort some function or procedure, or your program, when any error occurs. Of course you can only EXIT to leave a function or procedure which is currently active, so you will probably want to have around some BOOLEAN variables to keep track of whether or not you are currently within routines which you might wish to EXIT from ERRORTRAP.

3. Remember to set XEQERR back to zero before leaving ERRORTRAP, otherwise your next error will abort to AMOS without calling ERRORTRAP.

14.10.3 XERRORTRAP

When STDERRORTRAP is called by entering a Control-C, it is possible to request a backtrace of suspended functions and procedures. This backtrace begins with the caller of the caller of STDERRORTRAP, which is usually the caller of ERRORTRAP, and hence the routine which was suspended. Thus, should STDERRORTRAP be called by a function or procedure local to your ERRORTRAP procedure, the backtrace will begin in the wrong place. This can be corrected by using XERRORTRAP, which takes a copy of the system INFOREC as its argument. It is used as follows:

```
PROCEDURE ERRORTRAP;
VAR   INFOP: ^INFOREC; INFO: INFOREC;
  PROCEDURE P1;
     BEGIN
        XERRORTRAP(INFO);
     END;
BEGIN
  INFOP := ERRORINFO;
  INFO := INFOP^;
  P1;
  INFOP^.XEQERR := INFO.XEQERR;
END;
```

Using XERRORTRAP, the backtrace will be displayed beginning with the caller of the routine which invokes ERRORINFO, thus producing a correct backtrace, even when called from an inner procedure. Begin at the caller of the procedure which set X to ERRORINFO.

14.10.4 ERROR

The procedure ERROR(x) takes an INTEGER x as argument and generates the corresponding system error. See the previous section for the list of error codes.

CHAPTER 15

ASSEMBLY LANGUAGE SUBROUTINES

Assembly language subroutines are assembly language programs that are callable by your AlphaPascal programs.

Why would you want to call assembly languages routines from a Pascal program? There are at least two good reasons. Firstly, not all the capabilities of the operating system (AMOS) have been directly included in AlphaPascal. The ability to write assembly language subroutines allows you to enrich AlphaPascal, as need requires, with additional capabilities. Secondly, routines written in assembly language execute significantly faster than routines written in Pascal. Thus, you may wish to identify key functions and procedures which are bottle necks in your programs, and rewrite them in assembly language.

This chapter describes how to write and use assembly language subroutines. It will be assumed in this chapter that you are an experienced assembly language programmer on the AMOS system. For more information on assembly language programming, please refer to the AMOS Assembly Language Programmer's Reference Manual (DWM-00100-43), the WD16 Microcomputer Manual (DWM-00100-04), and the AMOS Monitor Calls Manual (DWM-00100-42).

15.1 CALLING ASSEMBLY LANGUAGE SUBROUTINES

In AlphaPascal, there is no distinction between calling an assembly language function or procedure, and calling a Pascal function or procedure which occurs in a separately compiled module. (Modules were discussed in Section 5.1.) Section 4.4.4 describes how to link an assembly language subroutine into a program during the PLINK process. Instead of linking output files from the compiler, you link a PRG file with an extension of .PSB which contains code for a single function or procedure. The name of the PSB file must be the first six letters of the name to be used for calling the assembly language routine. When specifying the file to PLINK you must, of course, specify the full name of the procedure or function contained in the PSB file, otherwise PLINK would not know the full name you wish to use for it.

For example, if you code in assembly language a procedure that displays a menu, the procedure name might be MENUDISP. The disk file containing that routine must then be called MENUDI. When you specify the file to PLINK, though, you use the full eight-character name of the procedure. For example:

```
File 1 = MENUDISP.PSB/LINK
```

15.2 ARGUMENT PASSING CONVENTIONS

Your assembly language routine must work with two stacks. One of these stacks is the familar SP stack. The other is a data stack used by AlphaPascal for passing arguments and recieving results. The data stack is indexed by R5, and so will also be referred to as the R5 stack. All other registers (R0-R4) are available for any purpose to your assembly routine.

Arguments are placed on the R5 stack in reverse order. That is, the last argument appears on the top of the R5 stack. For example, if we have the following program in AlphaPascal:

File TEST1.PAS--

```
PROGRAM;
EXTERNAL PROCEDURE demo1(x,y: INTEGER);
BEGIN
        demo1(10,20);
END.
```

then, upon entry to our assembly language subroutine, 20 will be on the top of the R5 stack (referenced as @R5) and 10 will be under it on the R5 stack (referenced as 2(R5)).

A procedure to print its two INTEGER arguments in order might then be DEMO1.MAC:

File DEMO1.MAC--

```
        COPY    SYS

START:  MOV     2(R5),R1        ; Get first argument
        DCVT    0,2             ; Print it in decimal
        CRLF
        MOV     @R5,R1          ; Get second argument
        DCVT    0,2             ; Print it in decimal
        CRLF
        ADDI    4,R5            ; Remove arguments from R5 stack
        RTN                     ; Return to pascal
```

DEMO1 would then be assembled with MACRO, and the resulting program file DEMO1.PRG would be renamed to DEMO1.PSB in order to allow it to be linked into a code file by PLINK.

When called as a function rather than as a procedure, your routine will receive an additional three words containing zeros on the top of the R5 stack. These words serve no purpose when writing assembly routines and may be immediately removed by executing an ADDI 6,R5. Their presence is required for internal reasons by functions written in Pascal.

Assembly language functions return their result on the top of the R5 stack after all arguments have been removed. Example:

File TEST2.PAS--

```
PROGRAM;
EXTERNAL FUNCTION Maximum(x,y: INTEGER): INTEGER;
BEGIN
   WRITELN(Maximum(2,7));
END.
```

File MAXIMU.MAC--

```
START:  ADDI    6,R5            ; Throw away unused additional words
        MOV     (R5)+,R2        ; Get 2nd argument
        MOV     (R5)+,R1        ; Get 1st argument
        CMP     R1,R2           ; 1st > 2nd ?
        BHI     USE1ST          ;  Yes, return 1st argument
        MOV     R2,-(R5)        ;  No, return 2nd argument
        RTN

USE1ST: MOV     R1,-(R5)        ; Return 1st argument
        RTN
```

After producing MAXIMU.PSB, you would need to remember to refer to the file as MAXIMUM.PSB to PLINK, otherwise it would think the function being defined had the name 'Maximu' instead of 'Maximum'.

15.2.1 Argument passing

There are two methods of passing arguments in Pascal, typified by:

```
      1.    PROCEDURE(x: INTEGER);
and   2.    PROCEDURE(VAR x: INTEGER);
```

In the first declaration, x is referred to as a value parameter. In the second declaration, x is referred to as a reference parameter.

In general, value parameters appear directly on the R5 stack, while reference parameters (denoting variables which can be modified) appear as an address on the R5 stack which points to the parameter.

However, there are exceptions: arrays, records, and strings always have their address passed on the R5 stack, even when they appear as value parameters.

15.2.2 Data Formats

This section describes the internal format of each data type. All data types are aligned on a word boundary unless contained as a packed field.

15.2.2.1 CHAR - Characters are represented by their ASCII code in a full machine word. They are only stored within single bytes of memory when contained in packed arrays or records.

15.2.2.2 INTEGER - Integers are represented in a single machine word.

15.2.2.3 BOOLEAN - Booleans are represented by a zero (FALSE) or one (TRUE) in a full machine word. They are only stored as single bits when contained in packed arrays or records.

15.2.2.4 Subranges and Scalar types - These are represented in a full machine word unless they appear in a packed array or record, in which case they are stored in a field of as many bits as necessary to hold their maximum value. User scalar types are numbered starting from zero.

15.2.2.5 REAL - Reals occupy three words of memory and conform with the format for reals used by the FADD, FSUB, FMUL, FDIV, and FCMP machine instructions.

15.2.2.6 STRING - Strings are represented by a length byte containing 0-255, followed by a sequence of bytes which are used to hold the actual characters of the string.

15.2.2.7 Pointers - Pointers require a full machine word.

15.2.2.8 Sets - Sets require one or more words depending upon the size of the set. Sets are represented as a bit pattern, where a one bit denotes the presence of a set element. The bits are ordered from low order to higher order in each word, and from first word to last word. For example, SET OF 3..19 requires two words of memory. The first three bits corresponding to 0 thru 2 are unused. To test for the presence of the element 18, one would perform a bit test on the second word of the set with a mask of 4.

15.2.2.9 Arrays - Arrays require one or more words. The elements of an array appear in order in memory. In packed arrays, the elements of an array may each occupy on a few bits, otherwise each element will appear on a word boundary. Fields appear from low order to high order in a word, and may not cross word boundaries.

15.2.2.10 Records - Records require one or more words. The elements of a record appear in order in memory in a fashion similar to arrays.

15.2.2.11 Files - Files are actually an internal kind of record format. The details of this format are not being made available as they will change as versions of AlphaPascal change.

15.2.3 Error Exit

Should you wish to generate an error from your assembly language subroutine, it is preferable that you call the Pascal system's ERRORTRAP procedure, rather than display an error and exit to AMOS directly, otherwise there is no guarantee that open files will be closed correctly.

To signal an error, you must perform a proper return from your routine, but in addition, advance your return address by executing IW2 @SP, and leave an execution error code in R1. For additional information on ERRORTRAP and a list of execution error codes, see section 14.10, "Error Handling Procedures and Variables."

15.3 CODE RESIDENCY

This section discusses the variety of ways in which your routine may appear in memory.

15.3.1 Routines PLINKed with /LINK

Routines which have been linked into a code file with the /LINK option must have a final PSB file which is exactly one block in size. Such routines are dynamically paged into memory along with Pascal psuedo-code. They are deleted from memory and reloaded as memory requirements and usage demand. They place no burden on available memory when not being used.

15.3.2 Routines PLINKed without /LINK

Routines which have been linked into a code file without the /LINK option will be searched for in memory and on disk each time they are called. What has been linked into the code file is not the actual routine, but rather the name of the PSB file containing that routine (see section 4.4.4).

If your routine has been loaded before entering AlphaPascal via the LOAD command, either into system memory or user memory, then that copy of your routine will be used.

If your routine is not present in memory, it will be temporarily loaded in order for it to be executed, and then deleted from memory immediately after execution.

15.4 OBTAINING MEMORY FOR DATA AREAS

When writing an assembly language routine, you will probably want and need temporary data areas. There is no room for allocating memory modules for this purpose. Instead, you may either allocate space for data in the SP stack, or place your data inline in your routine (this is unacceptable for routines which are to be loaded into system memory, since they must be sharable). The R5 stack is NOT available for allocating data space.

Another method for obtaining larger data areas, is to have your caller pass them to you as arguments.

15.5 RESTRICTIONS

As mentioned above, there is no room for allocating memory modules. This also means that you may not use INIT to create a file buffer, or perform file operations which would require loading a device driver into memory.

CHAPTER 16

WRITING AND MODIFYING AN EXTERNAL LIBRARY

When you link together your programs using PLINK, you are asked to specify a library file. Typically, you specify STDLIB. The global functions, procedures, and variables contained in this library are available to you just as if you wrote them in a module and linked them into your program. However, using routines contained in a library requires no additional space in your program's code file because the routines are accessed directly from the library file at run-time.

There are several advantages to placing commonly used routines in a library rather than linking them directly into your program. First, you save disk space by only having a single copy of your routines on disk. Second, the linking process is faster if you only need to specify a library rather than several files contain your modules. Finally, if it becomes necessary to modify a routine, you need only change it in the library to update all your programs which use it.

Another possible use of libraries is to generate multiple configurations of a program. A single program could be linked to a variety of libraries each of which define the same set of functions and procedures, but each of which do so with different definitions. This might be used to configure a generalized set of applications programs for use in different specific applications.

It is not necessary to specify EXTERNAL declarations for most of the functions and procedures in STDLIB. This is NOT a feature of libraries. Rather, the compiler has been written to automatically include EXTERNAL declarations for these commonly used routines.

There is really very little difference between a program file and a library file. Both are actually AlphaPascal programs. The only difference is that if Program A uses Program B as a library, then Program B is executed with the purpose of initializing the library (i.e., global variables in the library), before Program A is executed.

It is possible for a library to itself have a library. Thus Program A can use Program B as a library, and Program B can use Program C as a library, in which case C, then B, and finally A are executed.

To allow programs to be written which can serve either directly as a program, or indirectly as a library, a special BOOLEAN function is provided, called MAINPROG, which takes no arguments and returns true if the program in which it is executing is being used as the main program, and false if the program in which it is executing is being used as a library to another program. The idea is to write a program in such a way that if it is being used as a library, all it does is initialize global variables.

16.1 STDLIB

STDLIB is a special library which itself has no library. It provides a basic set of mandatory procedure and function definitions. It is permissable for you to overide any of these definitions with your own external procedures or functions with the exception of RDC, RDI, RDR, RDS, RLN, WLN, WRB, WRC, WRI, WRR, and WRS. Calls to these procedures are automatically generated whenever you use READ and WRITE statements. READ and WRITE will seriously malfunction if you redefine any of these.

The functions and procedures included in STDLIB are:

ARCCOS	Arc cosine function
ARCCOSH	Hyperbolic arc cosine function
ARCSIN	Arc sine function
ARCSINH	Hyperbolic arc sine function
ARCTAN	Arc tangent function
ARCTANH	Hyperbolic arc tangent function
CONCAT	Function to concatenate strings
COPY	Function to copy characters in string
COS	Cosine function
COSH	Hyperbolic cosine function
DELETE	Procedure to delete characters in string
ERRORTRAP	Default error handler
EXP	Function to compute e to the specified power.
FACTORIAL	Factorial function (X!)
GETFILE	Procedure to get information in filespec
GETLOCKS	Procedure to read file locks.
INCHARMODE	Returns true if terminal is in Charmode.
INSERT	Procedure to insert characters into a string
KILCMD	Procedure to abort command file
LCS	Function to convert upper case characters to lower case
LN	Function to compute natural (Napierian) log
LOG	Function to compute log base ten of argument
OPEN	Procedure to open an AMOS file
POS	Function to compute position of character in string
POWER	POWER(x,y) computes x to the y'th power
PROGRAM	STDLIB initialization
PWROFTWO	Function to compute powers of two
RDC	Routine used by READ
RDI	Routine used by READ
RDR	Routine used by READ
RDS	Routine used by READ

RESET	Procedure to close a file, and then open for input
REWRITE	Procedure to close, erase, and then open a file for output
RLN	Routine used by READ
SETFILE	Procedure to place file information in filespec
SIN	Sine function
SINH	Hyperbolic sine function
SPL	Routine used by SPOOL; must not be called directly
SPOOL	Procedure to spool files to line printer
STDERRORTRAP	Standard error handler
STRIP	Procedure to strip trailing blanks from string
SQRT	Square root function
TAN	Tangent function
TANH	Hyperbolic tangent function
TOD	Returns time of day in seconds as a real number
UCS	Procedure to convert upper case characters to lower case
WLN	Routine used by WRITE
WRB	Routine used by WRITE
WRC	Routine used by WRITE
WRI	Routine used by WRITE
WRR	Routine used by WRITE
WRS	Routine used by WRITE
XERRORTRAP	Special version of error handler
XLOCK	Procedure to set or release file locks
XMNT	Routine used by XMOUNT; must not be called directly
XMOUNT	Procedure to mount a disk

16.2 WRITING LIBRARY FILES

It is not likely that you would want to dispense with the standard library file altogether, since the compiler relies on the presence of many of the procedures and routines in that library. If you did not use STDLIB, you would have to duplicate for yourself all of the routines listed above that make up that library.

However, it is possible for one library to make use of another. For example, suppose you want to write your own library which contains a set of functions that are particularly useful for the programs that you write (e.g., you need a set of routines that construct and display screen menus), you can write such a library; then, when you link it, you can specify the STDLIB external library as its library file. (The only time you ever link a file without specifying a library, is when you are linking a root library, such as STDLIB itself-- a very rare occurrence.) In this case, your library file (perhaps named NEWLIB) would be linked with STDLIB. Then, when you link a new program, you might link it with the NEWLIB library. Your new program would thus be linked with NEWLIB which in turn has its own library, STDLIB. There is no limit to library nesting.

There are several things you should keep in mind when writing an external library:

1. If an external procedure or function is declared both in a program and in a library which it uses, then the definition within the program is in effect while execution resides in the program, and the definitions within its libraries are in effect while in its libraries.

2. If you change a procedure from pascal to assembly language, or from assembly language to pascal, it is wise to re-create (re-link) that program and all the programs which use it as a library. Any references to the procedure which are not re-linked will treat it as the wrong kind of code.

3. Similarly, if while updating a program, you overide a definition in a library which was formerly accessible, there is no garauntee that all references to the definition will be updated unless you re-link the program and all the programs which use it as a library.

4. If a library is updated with PLINK, it is not necessary to update the programs which use that library. However, if the library must be completely re-created, all programs which use that library will need to be re-created. Thus, it is desirable to avoid the need to re-create a library. PLINK does not allow you to enlarge the size of global variables with an update, thus it is wise to avoid having global variables which you may wish to enlarge, such as records, strings, or arrays. Instead use a global pointer variable which points the desired object. In this way, if you change the size of the object, no global variable will change size.

16.3 MODIFYING STDLIB

If you decide to modify STDLIB, you must do so very carefully. Because PLINK uses STDLIB while it is working, you must not directly modify STDLIB. If you want to add routines to STDLIB, use the AMOS COPY command to make a duplicate of STDLIB under another name. Then, add your routines to the copy of STDLIB using PLINK. Finally, rename your copy to STDLIB (making sure to keep a copy of the old STDLIB somewhere in case of emergencies).

However, it is far wiser to create a library which has STDLIB as its library, rather than to directly modify STDLIB. Otherwise, when Alpha Micro releases an update to STDLIB, all your programs will need to be re-linked!

16.4 VERSION CHECKING

Both PLINK and PRUN check to insure that a program is only given its original library or an update of that file, since an attempt to use any other file as a library results in a system crash.

If you attempt to execute a program with an improper library, you will receive the error

```
?Wrong version of xxx for use with yyy
```

where xxx is the name of your program and yyy is the name of its library. If you get this message, it either means that you are running with an out of date version of library yyy, or that you are running with a newer version of library yyy which had to be re-created. In the latter case, you will need to re-create your program with PLINK.

PART IV

APPENDICES

APPENDIX A

QUICK REFERENCE TO ALPHA PASCAL

This appendix gives a quick summary of the Pascal language as implemented by Alpha Micro. For information on a particular Pascal statement or element, look in the index to see what pages of this book contain information on that element. For a complete description of the standard Pascal language, see Jensen and Wirth, The Pascal User Manual and Report.

For a list of all standard identifiers, see Section 5.4.2, "Standard Identifiers."

A.1 PROGRAM STRUCTURE

A program consists of a heading and a block, and it concludes with a period:

```
Heading
     Block.
```

The heading takes this form:

```
PROGRAM program-name;
```

or:

```
PROGRAM;
```

A block has the form:

```
label declaration
constant definitions
type definitions
variable declarations
external declarations
procedure and function declarations
BEGIN statement1 ; statement2 ; ... ; statementN END.
```

If a file is not a main program file, the heading takes the form:

```
MODULE module-name;
```

or:

```
MODULE;
```

and the block takes the form:

```
label declaration
constant definitions
type definitions
variable declarations
external declarations
procedure and function declarations
```

A.2 DECLARATIONS AND DEFINITIONS

Pascal requires that you define and declare all variables, labels, constants, data types, procedures, and functions at the front of each program or procedure.

A.2.1 Label Declarations

Labels are always unsigned integers. A label declaration takes the form:

```
LABEL integer1, integer2, .... integerN ;
```

A.2.2 Constant Definitions

```
CONST     identifier1 = value1;
          identifier2 = value2;
                .
                .
          identifierN = valueN;
```

A.2.3 Type Definitions

```
TYPE      identifier1 = type1;
          identifier2 = (identifier3, identifier4, ...);
          identifier5 = value1..value2;
          ... ;
```

A.2.4 Variable Declarations

```
VAR     identifier ..., identifier : data-type;
        identifier ..., identifier : value1..value2;
        ... ;
```

A.2.5 Procedure Declarations

```
PROCEDURE procedure-name;
    block;
```

or:

```
PROCEDURE procedure-name(formal-parameters1;...formal-parametersN);
    block;
```

where formal-parameters have the form:

```
identifier1 ..., identifierN : type1
```

or:

```
VAR identifier1 ..., identifierN : type1
```

A.2.6 Function Declarations

```
FUNCTION function-name : result-type;
    block;
```

or:

```
FUNCTION function-name(formal-parameters1;
                       ...formal-parametersN) : result-type;
    block;
```

where formal-parameters have the form:

```
identifier1 ..., identifierN : type1
```

or:

```
VAR identifier1 ..., identifierN : type1
```

A.3 DATA TYPES

The data type tells Pascal what range of values the declared variable may assume and what operations may be carried out on those variables. Data types are simple data types or structured data types.

A.3.1 Simple Data Types

A simple data type is the basic data type of which structured data types are built. The simple data type is called a "scalar type." Such a type contains a set of elements, and those elements are ordered.

A.3.1.1 Standard Data Types - The standard data types are:

INTEGER - A non-fractional number in the range -32767 through 32767.

REAL - A floating point number significant to 11 digits (12 for integer values) with an exponent range of roughly 1E-37 to 1E37.

BOOLEAN - The standard scalar type (FALSE, TRUE).

CHAR - A single ASCII character.

A.3.1.2 User-defined Scalar Types - A scalar data type takes the form:

(identifier-element1, identifier-element2,...identifier-elementN)

or a subrange type (of another, already defined scalar type) of the form:

first-element .. last-element

A.3.2 Structured Data Types

Simple data types can be organized into larger units, called structured types. A type definition or variable declaration of a structured data type that includes the keyword PACKED tells the compiler to minimize internal storage for that data type (at the possible expense of execution time). For example, instead of:

```
VAR LongLine : ARRAY [1..1000] OF CHAR;
```

you could cause LongLine to be a packed array by saying:

```
VAR LongLine : PACKED ARRAY [1..1000] OF CHAR;
```

The structured data types are:

A.3.2.1 STRING - STRING data is a group of characters. You may optionally specify a maximum length by following the keyword STRING with square brackets enclosing the number (e.g., STRING[23]).

A.3.2.2 Arrays -

ARRAY [index1-type, index2-type, ..., indexN-type] OF component-type

A.3.2.3 Sets -

SET OF element-type

A.3.2.4 File Type -

FILE OF element-type

or:

TEXT

(This is the same as "FILE OF CHAR".)

A.3.2.5 Record Type -

RECORD field-list END

where field list is of the form:

```
field-identifier .., field-identifierN  : field1-type1;
field-identifier .., field-identifierN  : field2-type2;
                  ...
field-identifier .., field-identifierN  : fieldN-typeN;
```

The field list may also contain a variant-part, which implies that the information in that field may vary as to type. The variant-part takes this form:

```
CASE field-type OF
     case-label ..., case-label : (field-list1);
     case-label ..., case-label : (field-list2);
     ...
     case-label ..., case-label : (field-listN)
```

or:

```
CASE case-field-identifier : field-type OF
     case-label ..., case-label : (field-list1);
     case-label ..., case-label : (field-list2);
     ...
     case-label ..., case-label : (field-listN)
```

A.3.2.6 Pointer Data Types - The pointer enables Pascal to permit dynamic data structures by giving you a way to point to an element of such a structure. It takes the form:

```
^object-type
```

Pascal provides a standard constant NIL, which points to "nothing."

A.4 EXPRESSIONS

Expressions use operators to combine variables, constants, and function calls into larger units. This section gives information about each of these components of an expression.

A.4.1 Operators

Operators have precedence, which you can override by incuding parentheses in the expression. The unary operators are performed before all other operators; next the multiplying operators are performed, followed by the adding operators. Then, the relational operators are performed. Lastly, the Boolean operators are applied. If several operators in an expression have the same precedence, execution is performed from left to right.

A.4.1.1 Assignment -

```
:=
```

A.4.1.1.1 The Modifying Assignment Operators -

The modifying assignment operators are:

+=	Addition
-=	Subtraction
*=	Multiplication
/=	Division

A.4.1.2 Arithmetic Operators: -

+	(unary operator) Identity
-	(unary operator) Sign inversion
+	Addition
-	Subtraction
*	Multiplication
DIV	Integer number division
/	Real number division
MOD	Modulus

A.4.1.3 Relational Operators -

=	Equality
<>	Inequality
<	Less than
>	Greater than
<=	Less than or equal (or, set inclusion)
>=	Greater than or equal (or, set inclusion)
IN	Set membership

A.4.1.4 Logical Operators -

NOT	Negation
OR	Disjunction
AND	Conjunction

A.4.1.5 Set Operators -

+	Union
-	Set difference
*	Intersection

A.4.2 Constants

Constants may consist of:

Characters and strings of characters (in quotes)

TRUE and FALSE

MAXINT (which evaluates to the largest integer on the AMOS system, 32767).

Values of user-defined types

Integers

Decimal and exponential numbers - If a number contains a decimal point, at least one digit must appear to the left of the decimal point. The exponent in an exponential number is identified by the "E" symbol. For example: "34E-5" represents "0.000034".

A.4.3 Variables

A variable is a simple identifier, an indexed variable of the form:

array-variable [index1-expression,...indexN-expresion]

a referenced variable or file buffer variable of the form:

pointer-variable^

or:

file-variable^

or a field designator of the form:

record-variable . field-identifier

A.4.4 Function Calls

Function calls have the form:

function-identifier

or:

function-identifier (parameter, ..., parameterN)

A.4.5 IF-THEN-ELSE and CASE-OF Constructs in Expressions

AlphaPascal allows you to use the IF-THEN-ELSE and CASE-OF constructs to conditionally evaluate one of two (in the case of the IF-THEN-ELSE) or several (in the case of the CASE-OF) expressions:

```
IF Boolean expression THEN expression ELSE expression
```

and:

```
CASE value OF
                value1 : expression;
                value2 : expression;
                    ...
                valueN : expression;
              ELSE expression;
```

A.5 STATEMENTS

Statements are either simple statements or structured statements. A simple statement consists of only one statement. Structured statements are comprised of more than one statement.

You may label statements by writing:

```
label: statement
```

where "label" is an unsigned integer.

A.5.1 Simple Statements

The Pascal simple statements are:

A.5.1.1 Assignment Statement - assigns a value to a variable:

```
variable := expression
```

A.5.1.2 Procedure Call - Procedure calls invoke the specified procedure, and take the form:

```
procedure-name
```

or:

```
procedure (parameter1, parameter2, ..., parameterN)
```

A.5.1.3 GOTO Statement - The GOTO statement transfers program control to the labeled portion of the program. It takes the form:

```
GOTO label
```

A.5.1.4 Null Statement - Another permissible simple statement is the null statement (that is, no statement at all).

A.5.2 Structured Statements

The Pascal structured statements are:

A.5.2.1 Compound Statements - The compound statement is bracketed with the keywords BEGIN and END, and takes the form:

```
BEGIN statement1; statement2; ...; statementN END.
```

A compound statement may take the place of any single statement in the examples given in this appendix.

A.5.2.2 Conditional Statements -
A conditional statement contains statements whose execution depends on the result of a conditional test. These statements may take the form:

```
IF Boolean expression THEN statement;
```

or:

```
IF Boolean expression THEN statement ELSE statement;
```

or:

```
CASE expression OF
     case1-label: statement1;
     case2-label: statement2;
     ...
     caseN-label: statementN
END.
```

(Several case-labels, separated by commas, may be written in place of a single case-label.)

A.5.2.3 Repetitive Statements -

```
WHILE Boolean expression DO statement
```

or:

```
REPEAT statement-list UNTIL Boolean expression
```

or:

```
FOR variable-identifier := expression TO expression
    DO statement
```

or:

```
FOR variable-identifier := expression DOWNTO expression
    DO statement
```

A.5.2.4 WITH Statement -

The WITH-DO statement allows you to access record fields as if they were simple variables:

```
WITH record-variable1, record-variable2, ..., record-variableN
    DO statement
```

A.6 ALPHA PASCAL STANDARD FUNCTIONS AND PROCEDURES

Below is an alphabetic list of all AlphaPascal standard functions and procedures that you may use. To find out what pages of this book discuss a particular procedure or function, refer to the Index.

ABS	ARCCOS	ARCCOSH	ARCSIN
ARCSINH	ARCTAN	ARCTANH	CHARMODE
CHR	CLOSE	CONCAT	COPY
COS	COSH	CREATE	CRT
DELETE	EOF	EOLN	ERASE
ERROR	ERRORINFO	ERRORTRAP	EXIT
EXP	EXPONENT	EXTENSION	FACTORIAL
FILLCHAR	FILESIZE	FSPEC	GET
GETFILE	GETLOCKS	INCHARMODE	INSERT
JOBDEV	JOBUSER	KILCMD	LCS
LENGTH	LINEMODE	LN	LOCATION
LOG	LOOKUP	MAINPROG	MARK
MEMAVAIL	MOVELEFT	MOVERIGHT	NEW
ODD	OPEN	OPENI	OPENO
OPENR	ORD	PAGE	PFILE
POS	POWER	PRED	PUT
PVIRT	PWROFTEN	PWROFTWO	RAD50
RANDOMIZE	READ	READLN	RELEASE
RENAME	RESET	REWRITE	RND
ROUND	SCAN	SEEK	SETFILE
SHIFT	SIN	SINH	SIZEOF
SPOOL	SQR	SQRT	STDERRORTRAP
STR	STRIP	SUCC	TAN
TANH	TIME	TOD	TRUNC
UCS	VAL	WRITE	WRITELN
XEQERR	XLOCK	XMOUNT	

For a list of all standard identifiers and reserved words, see Section 5.4, "Legal Identifiers."

A.5.2.3 Repetitive Statements -

```
WHILE Boolean expression DO statement
```

or:

```
REPEAT statement-list UNTIL Boolean expression
```

or:

```
FOR variable-identifier := expression TO expression
   DO statement
```

or:

```
FOR variable-identifier := expression DOWNTO expression
   DO statement
```

A.5.2.4 WITH Statement -
The WITH-DO statement allows you to access record fields as if they were simple variables:

```
WITH record-variable1, record-variable2, ..., record-variableN
   DO statement
```

A.6 ALPHA PASCAL STANDARD FUNCTIONS AND PROCEDURES

Below is an alphabetic list of all AlphaPascal standard functions and procedures that you may use. To find out what pages of this book discuss a particular procedure or function, refer to the Index.

ABS	ARCCOS	ARCCOSH	ARCSIN
ARCSINH	ARCTAN	ARCTANH	CHARMODE
CHR	CLOSE	CONCAT	COPY
COS	COSH	CREATE	CRT
DELETE	EOF	EOLN	ERASE
ERROR	ERRORINFO	ERRORTRAP	EXIT
EXP	EXPONENT	EXTENSION	FACTORIAL
FILLCHAR	FILESIZE	FSPEC	GET
GETFILE	GETLOCKS	INCHARMODE	INSERT
JOBDEV	JOBUSER	KILCMD	LCS
LENGTH	LINEMODE	LN	LOCATION
LOG	LOOKUP	MAINPROG	MARK
MEMAVAIL	MOVELEFT	MOVERIGHT	NEW
ODD	OPEN	OPENI	OPENO
OPENR	ORD	PAGE	PFILE
POS	POWER	PRED	PUT
PVIRT	PWROFTEN	PWROFTWO	RAD50
READ	READLN	RELEASE	RENAME
RESET	REWRITE	ROUND	SCAN
SEEK	SETFILE	SHIFT	SIN
SINH	SIZEOF	SPOOL	SQR
SQRT	STDERRORTRAP	STRIP	SUCC
TAN	TANH	TIME	TOD
TRUNC	UCS	WRITE	WRITELN
XEQERR	XLOCK	XMOUNT	

For a list of all standard identifiers and reserved words, see Section 5.4, "Legal Identifiers."

APPENDIX B

THE ASCII CHARACTER SET

The next few pages contain charts that list the complete ASCII character set. We provide the octal, decimal and hexadecimal representations of the ASCII values.

Note that the first 32 characters are non-printing Control-characters.

THE CONTROL CHARACTERS

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
NULL	000	0	00	Null (fill character)
SOH	001	1	01	Start of Heading
STX	002	2	02	Start of Text
ETX	003	3	03	End of Text
ECT	004	4	04	End of Transmission
ENQ	005	5	05	Enquiry
ACK	006	6	06	Acknowledge
BEL	007	7	07	Bell code
BS	010	8	08	Back Space
HT	011	9	09	Horizontal Tab
LF	012	10	0A	Line Feed
VT	013	11	0B	Vertical Tab
FF	014	12	0C	Form Feed
CR	015	13	0D	Carriage Return
SO	016	14	0E	Shift Out
SI	017	15	0F	Shift In
DLE	020	16	10	Data Link Escape
DC1	021	17	11	Device Control 1
DC2	022	18	12	Device Control 2
DC3	023	19	13	Device Control 3
DC4	024	20	14	Device Control 4
NAK	025	21	15	Negative Acknowledge
SYN	026	22	16	Synchronous Idle
ETB	027	23	17	End of Transmission Blocks
CAN	030	24	18	Cancel
EM	031	25	19	End of Medium
SS	032	26	1A	Special Sequence
ESC	033	27	1B	Escape
FS	034	28	1C	File Separator
GS	035	29	1D	Group Separator
RS	036	30	1E	Record Separator
US	037	31	1F	Unit Separator

PRINTING CHARACTERS

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
SP	040	32	20	Space
!	041	33	21	Exclamation Mark
"	042	34	22	Quotation Mark
#	043	35	23	Number Sign
$	044	36	24	Dollar Sign
%	045	37	25	Percent Sign
&	046	38	26	Ampersand
'	047	39	27	Apostrophe
(	050	40	28	Opening Parenthesis
)	051	41	29	Closing Parenthesis
*	052	42	2A	Asterisk
+	053	43	2B	Plus
,	054	44	2C	Comma
-	055	45	2D	Hyphen or Minus
.	056	46	2E	Period
/	057	47	2F	Slash
0	060	48	30	Zero
1	061	49	31	One
2	062	50	32	Two
3	063	51	33	Three
4	064	52	34	Four
5	065	53	35	Five
6	066	54	36	Six
7	067	55	37	Seven
8	070	56	38	Eight
9	071	57	39	Nine
:	072	58	3A	Colon
;	073	59	3B	Semicolon
<	074	60	3C	Less Than
=	075	61	3D	Sign
>	076	62	3E	Than
?	077	63	3F	Question Mark
@	100	64	40	Commercial At

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
A	101	65	41	Upper Case Letter
B	102	66	42	Upper Case Letter
C	103	67	43	Upper Case Letter
D	104	68	44	Upper Case Letter
E	105	69	45	Upper Case Letter
F	106	70	46	Upper Case Letter
G	107	71	47	Upper Case Letter
H	110	72	48	Upper Case Letter
I	111	73	49	Upper Case Letter
J	112	74	4A	Upper Case Letter
K	113	75	4B	Upper Case Letter
L	114	76	4C	Upper Case Letter
M	115	77	4D	Upper Case Letter
N	116	78	4E	Upper Case Letter
O	117	79	4F	Upper Case Letter
P	120	80	50	Upper Case Letter
Q	121	81	51	Upper Case Letter
R	122	82	52	Upper Case Letter
S	123	83	53	Upper Case Letter
T	124	84	54	Upper Case Letter
U	125	85	55	Upper Case Letter
V	126	86	56	Upper Case Letter
W	127	87	57	Upper Case Letter
X	130	88	58	Upper Case Letter
Y	131	89	59	Upper Case Letter
Z	132	90	5A	Upper Case Letter
[	133	91	5B	Opening Bracket
\	134	92	5C	Back Slash
]	135	93	5D	Closing Bracket
^	136	94	5E	Circumflex
_	137	95	5F	Underline
`	140	96	60	Grave Accent
a	141	97	61	Lower Case Letter
b	142	98	62	Lower Case Letter
c	143	99	63	Lower Case Letter
d	144	100	64	Lower Case Letter
e	145	101	65	Lower Case Letter
f	146	102	66	Lower Case Letter
g	147	103	67	Lower Case Letter
h	150	104	68	Lower Case Letter
i	151	105	69	Lower Case Letter
j	152	106	6A	Lower Case Letter
k	153	107	6B	Lower Case Letter
l	154	108	6C	Lower Case Letter
m	155	109	6D	Lower Case Letter
n	156	110	6E	Lower Case Letter
o	157	111	6F	Lower Case Letter

CHARACTER	OCTAL	DECIMAL	HEX	MEANING
p	160	112	70	Lower Case Letter
q	161	113	71	Lower Case Letter
r	162	114	72	Lower Case Letter
s	163	115	73	Lower Case Letter
t	164	116	74	Lower Case Letter
u	165	117	75	Lower Case Letter
v	166	118	76	Lower Case Letter
w	167	119	77	Lower Case Letter
x	170	120	78	Lower Case Letter
y	171	121	79	Lower Case Letter
z	172	122	7A	Lower Case Letter
{	173	123	7B	Opening Brace
|	174	124	7C	Vertical Line
}	175	125	7D	Closing Brace
~	176	126	7E	Tilde
DEL	177	127	7F	Delete

APPENDIX C

ALPHA PASCAL COMPILER ERROR MESSAGES

Below is an alphabetic list of all error messages output by the AlphaPASCAL compiler. For a discussion of how to compile programs, and for information on error reporting and error recovery, see Chapter 4, "Operating Instructions and Characteristics."

We believe that the error messages below are very helpful in explaining exactly what part of your program caused the error. Therefore we have not provided detailed explanations for each error message. For some of the messages below we have added notes that give more information about the error and that tell you where to look in this manual for more information on the operator, data structure, or declaration involved in the error.

When CMPILR displays an error message, it also displays the line of the program that contains the error and points to the problem. For example, if you try to compile the following small program:

```
PROGRAM TestError;

VAR     Number1 : REAL;
        Number2 : STRING;

BEGIN { Try to use addition operator on real and string data. }
    IF Number1 + Number2 = 0 THEN WRITELN('Zero.')
END.
```

you see the following display:

```
AlphaPascal Compiler Version 2.0
           <   0>-----
 PROGRAM   <   5>-
BEGIN { Try to use addition operator on real and string data. }
    IF Number1 + Number2 = 0 THEN WRITELN('Zero.')
                         ^

?Line 6: [BOPNBN] In 'x+y', x and y are not both numeric
?HIT RETURN to continue
<   6>-
 7 lines
 4.10 seconds,  102.44 lines/minute
?Total of 1 compilation errors.
```

The error above occurred because we tried to perform an arithmetic operation on numeric and string data; both Number1 and Number2 must be numeric in order to use the addition operator.

The first eight characters of the error message identify the portion of the compiler that caught the error. You will probably not need to make note of this identifier.

In many of the error messages, CMPILR actually substitutes into the error message the operator or identifier that is the source of the error. For example, in the list below, the error message above appears as:

```
[BOPNBN] In 'x <op> y', x and y are not both numeric
```

In our example above, CMPILR substituted into the error message the operator ("<op>") causing the problem, and displayed the message:

```
[BOPNBN] In 'x+y', x and y are not both numeric
```

The symbols in the error messages that are replaced by elements from your program when the message is displayed are:

<op>	Operator
XXX	User-defined Identifier
xxx yyy zzz	Keyword

C.1 THE ERROR MESSAGES

[??????] *** Undefined error ***
You should never see this error message. Please report it and the circumstances under which you saw it to Alpha Micro.

[ANEANX] In 'x AND y', x must be of type BOOLEAN
[ANENOT] In 'NOT x', x must be of type BOOLEAN
See Chapter 8 for information on BOOLEAN operators.

[ASGAST] In 'x:=y', the types of x and y are incompatible

[ASGFIL] It is illegal to assign files to one another
See Chapter 7 for information on the FILE data type.

[ASGMAT] In 'x <op>= y', the types of x and y are incompatible
You tried to use a modifying assignment operator on two pieces of data that were of incompatible type. For example, you cannot use "NUMBER /= DATA" if NUMBER is an INTEGER but DATA is REAL, since you cannot return an INTEGER result if you divide an INTEGER by a REAL number.

[ASGSWL] String constant has wrong length for packed array
[BDYULB] Undefined labels occur in this function/procedure

[BEXARL] Only '=' and '<>' are permitted with ARRAYs
See Chapter 7 for information on ARRAY data types.

[BEXCMT] In 'x <relation> y', x and y are incompatible
[BEXFRL] Comparison of FILEs is undefined
[BEXINS] In 'x IN y', y must be a SET type
[BEXINT] In 'x IN y', x must be compatible with base type of y
[BEXPRL] Only '=' and '<>' are permitted with pointers
[BEXRRL] Only '=' and '<>' are permitted with RECORDs
[BEXSRL] '<' and '>' are undefined on SETs

[BLKDOT] '.' (denoting end of source) expected - assumed missing
CMPILR reached the end of the file, but saw no period. Remember to end all program and module files with a period.

[BOPINT] Only INTEGER operands are permitted with <op>
[BOPIOS] Only INTEGER or set operands are permitted with <op>
[BOPNBN] In 'x <op> y', x and y are not both numeric
[BOPNBS] In 'x <op> y', x and y are not both sets
[BOPNOS] Only numeric or set operands are permitted with <op>
[BOPNUM] Only numeric (INTEGER or REAL) operands are permitted with <op>
[CALAPB] Preceding argument must not be a packed char field
[CALARL] The preceding string constant has wrong length
[CALARS] The preceding SET variable has wrong size
[CALART] The preceding argument has wrong type
[CALARV] The preceding argument must be a variable expression
[CALCHR] The preceding must be of type CHAR

[CALEXT] EXIT(x) where x is a standard func or proc is illegal
You may only supply EXIT with the PROGRAM keyword or the name of your own procedure or function that you want to exit; you may not supply the name of a function or procedure in the library.

[CALFIL] Preceding argument must be of FILE type
[CALFRM] Formal procedures and functions not implemented
[CALINT] The preceding argument must be of type INTEGER
[CALIOR] Preceding argument must be of type INTEGER or REAL
[CALLPR] '(' expected -- assumed missing
[CALNRS] Preceding argument must be a pointer or non-REAL scalar

[CALOPM] INPUT, OUTPUT,or RANDOM expected -- INPUT assumed
See Chapter 10 for information on file-identifiers.

[CALPAC] Must be a packed array of char or a char element
[CALPTV] The preceding must be a pointer variable
[CALRDP] It is illegal to read into a packed char field
[CALRDT] Arguments to read must be INTEGER, REAL, CHAR, or String
[CALSCN] Only '=' and '<>' are permitted here
[CALSEX] The preceding must be a string expression
[CALSVR] The preceding must be a string variable
[CALTFA] Too few arguments supplied
[CALTGS] The preceding must not be a string or a real

[CALTGT] The preceding constant is of incorrect type for variant
See Chapter 7 for information on RECORD variants.

[CALTMA] Too many arguments supplied
[CALTXT] Preceding argument must be of type TEXT (FILE OF CHAR)
[CALWRM] Preceding modifier must be of type INTEGER
[CALWRT] Must be INTEGER, REAL, CHAR, String, or pck'd arry of chr
[CSDJNK] Junk after <constant definition> -- scanning
[CSTSGN] Only INTEGER and REAL constants may be signed
[EXPORX] In 'x OR y', x must be of type BOOLEAN
[FACCET] In 'CASE x OF ...', x must be a non-REAL scalar type
[FACCLT] In 'CASE x OF ...', labels must be compatible with x
[FACCVT] In CASE expressions, all cases must have compatible types
[FACDCS] The previous case label has already appeared
[FACIFT] THEN and ELSE expressions must have compatible types
[FACRTL] Proc or func too large, split it into smaller pieces
[FACSCK] In set constructor [--], set elements must be scalars
[FACSCT] In set constructor [--], all elements must be compatible
[GVDFIL] Global files must be declared in PROGRAM file
[GVDFWP] ^x present and x never declared for some x
[GVDJNK] Junk after <variable definition> -- scanning
[INIESF] Empty source file
[INILTL] First source line too long -- truncated to 132 characters
[INIPOM] PROGRAM or MODULE expected -- 'PROGRAM;' assumed
[INIRPR] ')' expected -- inserting ')'
[INISEM] ';' expected -- inserting ';'
[INISOI] ';' or <identifier> expected -- inserting ';'
[INISOP] ';' or '(' expected -- inserting ';'

(Changed 30 April 1981)

[LADERR] *** Compiler error in LOADADDRESS ***
You should never see this error message. Please report it and the circumstances under which you saw it to Alpha Micro.

[LADPCK] Packed variables may not be used in this context
[LBLDDC] Label already declared in this scope
[MPATBG] Maximum string size is 255
[PRDAFL] Only formal (VAR) FILE parameters are permitted
[PRDDDF] Function or procedure already declared forward
[PRDDDP] Parameter-list must only appear in FORWARD declaration
[PRDFNR] ': <result type identifier>' expected -- assumed missing

[PRDFTM] Function type not compatible with forward declaration
For information on forward declarations, see Chapter 6.

[PRDLPX] '(' expected -- assumed missing
[PRDNST] Procedure/function declarations nested too deeply
[PRDPDF] Function or procedure previously defined
[PRDPRF] Previously declared a function in same scope
[PRDPRP] Previously declared a procedure in same scope
[PRDSSP] Function must be of scalar, subrange, or pointer type
[SCNINS] Giving up scan -- inserting xxx
[SCNMIS] Giving up scan -- xxx assumed missing
[SELATO] In x[y] or x[--,y], x or x[--] must be of ARRAY type

[SELERR] *** Compiler error in SELECT ***
You should never see this error message. Please report it and the circumstances under which you saw it to Alpha Micro.

[SELFOP] In 'x^', x must be of pointer or FILE type
[SELIXT] In x[y], y must be compatible with index type of x
[SELNIS] Only enclosing func identifiers may be used as variables
[SELNSF] In 'x.y', y must be a field of the RECORD x
[SELRTO] In 'x.y', x must be of RECORD type

[SELSTF] Standard function identifiers may not be used as variables
For a list of the standard identifiers, see Chapter 5.

[SELSXO] In x[y], y must be of non-REAL scalar type
[SIDUDF] 'XXX' is undefined
[SIDWRC] 'XXX' is not a TYPE/CONST/VAR/FIELD/PROCEDURE/FUNCTION identifier
[SMPNUM] In '-x', x must be numeric

[STMBID] Wrong BEGIN-END identifier -- XXX expected
For information on BEGIN-END labels, see Section 6.2, "Label Declarations."

[STMCSD] The preceding case label appears more than once
[STMCST] The preceding case label has wrong type
[STMDOW] DO without WHILE, FOR, or WITH
[STMEWI] ELSE without IF or CASE
[STMFFK] Final FOR value must be of scalar type
[STMFFT] FOR variable and final value have incompatible types

(Changed 30 April 1981)

[STMFIK] Initial FOR value must be of scalar type
[STMFIT] FOR variable and initial value have incompatible types
[STMFVF] In 'FOR x:=...', x must not be a formal variable
[STMFVK] In 'FOR x:=...', x must be a non-REAL scalar variable
[STMGTO] GOTO statements are not permitted without (*$G+*) option
[STMMDL] Definition for this label has already appeared
[STMPEX] Function calls are not legal as statements
[STMPEX] Procedure identifier was expected
[STMRTL] Proc or func too large, split it into smaller pieces
[STMTWI] THEN without IF
[STMULB] Undeclared label
[STMUWR] UNTIL <expression> without REPEAT

[STMWRT] In 'WITH x DO ...', x must be a RECORD variable
See Chapter 9 for information on accessing record fields with WITH-DO.

[STMWTS] WITH statement has caused too many nested scopes

[STRERR] *** Compiler error in STORE ***
You should never see this error message. Please report it and the circumstances under which you saw it to Alpha Micro.

[TOKEDG] Digit (0-9) expected in exponent -- assumed missing

[TOKEOF] Unexpected end-of-source-file encountered
Remember to end every program or module file with a period.

[TOKFDG] Digit (0-9) expected in fraction -- assumed missing
[TOKILC] Illegal character encountered -- ignoring

[TOKINF] Include file not found
See Chapter 4 for information on Include Files.

[TOKIRG] Integer constants must be in the range +-32767
[TOKLTL] Line too long -- truncated to 132 characters
[TOKNIN] File includes (*$I ---*) may not be nested
[TOKSLS] Unterminated string (multi-line strings not permitted)
[TRMNBN] In 'x/y', both operands must be numeric
[TRYINS] xxx or yyy expected -- inserting xxx
[TRYINS] xxx or yyy expected -- inserting yyy
[TRYINS] xxx, yyy, or zzz expected -- inserting xxx
[TRYINS] xxx, yyy, or zzz expected -- inserting yyy
[TRYMIS] xxx expected -- yyy assumed missing
[TRYSCN] xxx expected -- scanning
[TRYSCN] xxx or yyy expected -- scanning
[TRYSCN] xxx, yyy, or zzz expected -- scanning
[TYDFWP] ^x present and x never declared for some x
[TYDJNK] Junk after <type definition> -- scanning
[TYPBTF] In 'ARRAY [x] OF y', y must not be a FILE type
[TYPCTK] In 'CASE x OF ...', x must be a scalar type identifier
[TYPCTR] In 'CASE x OF ...', x must not be of type REAL
[TYPIXB] Array is too large

(Changed 30 April 1981)

[TYPIXR] In 'ARRAY [x] OF y', x must not be of type REAL
[TYPIXT] In 'ARRAY [x] OF y', x must be a scalar type
[TYPLGH] x..y where x>y is illegal
[TYPNST] Declarations too deeply nested
[TYPRFF] Record fields must not be of FILE type
[TYPRGE] x..y where x and y are incompatible
[TYPSCI] A string constant identifier must not appear here
[TYPSRR] Subranges of type real are illegal
[TYPSTB] Set is too large
[TYPSTB] Set is too large (must be <= SET OF 0..4095)
[TYPSTK] In 'SET OF x', x must be a scalar type
[TYPSTR] In 'SET OF x', x must not be of type REAL
[TYPSXR] STRING[x] must have 1 <= x <= 255
[TYPSXT] STRING[x] where x is not an integer
[TYPTTE] In 'CASE x OF ...', tag type is incompatible with x
[VRDFWP] ^x present and x never declared for some x
[VRDJNK] Junk after <variable definition> -- scanning
[XPRAFL] Only formal (VAR) FILE parameters are permitted
[XPRLPX] '(' expected -- assumed missing
[XPRSSP] Function must be of scalar, subrange, or pointer type

Index